1-1990

Design and implementation of a systolic array to solve the Algebraic Path Problem with the specific instance of the transistive and reflexive closure of a binary relation

David Gene McCall

Design and Implementation of a Systolic Array to Solve the Algebraic Path Problem with the Specific Instance of the Transistive and Reflexive Closure of a Binary Relation

by

David Gene McCall

A Thesis Submitted
in
Partial Fulfillment
of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Prof. George A. Brown
(Thesis Advisor)

Prof. _____

Prof. _____

Prof. Roy S. Czernikowski
(Department Head)

DEPARTMENT OF COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
MAY 1990

TITLE OF THESIS: The Design and Implementation of Systolic Array to Solve the Algebraic Path Problem with Specific Instance to the Transistive and Reflexive Closure of a Binary Relation.

I, David Gene McCall, hereby grant permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use of profit.

DATE 5/17/90 David McCall

Table of Contents

Table of Tables

Table of Figures

1.0 Abstract:

The Algebraic Path Problem (APP) has many practical instances to be solved. The general solution by Robert and Trystram (1986) will be discussed along with the mapping and operation of the algorithm to a systolic array. The specific instance of the APP, the transistive and reflexive closure of a binary relation, will be implemented with a discussion of the different stages ranging from the logic equations to a method of the fabrication.

2.0 Introduction:

A systolic array has been defined by Will Moore as "a regular array of processing elements all doing the same calculation and passing results on to their nearest neighbors every cycle." (Robert 1986) Although this definition is not strictly adhered to by many systolic architectures, it is still the basic underlying theme.

The systolic array that will be presented here is a regular array of three different processing elements performing similar functions. The interconnections are only to the closest horizontal or vertical neighbors. It will solve the general Algebraic Path Problem (APP).

The discussion will begin with the theory of the APP followed by the presentation of Yves Robert and Denis Trystram's (1986) algorithm for its solution. Their mapping of the algorithm and its operation as implemented on a systolic array will be explained and clarified.

A specific instance of the APP, the transistive and reflexive closure of a binary relation, will be selected and implemented. The complete stages will be stepped through from circuit

design and testing to the integrated circuit layout of MOSIS CMOS3 (MOSIS3) standard cells. At the end of the discussion a fabrication file will be completed allowing the chip to be fabricated.

3.0 Formal Description of Algebraic Path Problem:

The Algebraic Path Problem (APP) deals with the problem of finding the distances between all vertices of a weighted graph over a specified semiring. To begin the discussion several definitions will be given along with some corresponding examples to clarify the concepts. The first definition is that of a semigroup, the building block of a semiring.

Definition 1:

A pair $(G,(\cdot))$ is a semigroup if and only if:

a) G is a (nonvoid) set.

b) $G \times G \dashrightarrow (\cdot)\ G$.

c) $(\cdot)$ is associative.

d) there exists a unique neutral element $(e) \in G$ such that, $g\ (\cdot)\ (e) = (e)\ (\cdot)\ g = g$ for every $g \in G$.

e) the group is commutative iff:

$g_1\ (\cdot)\ g_2 = g_2\ (\cdot)\ g_1$ for all $g_1, g_2 \in G$.

(Hartnett 1963, p. 160)

The semigroup is equivalent to a moniod which has a variation on the notation, that is $(G,(\cdot)) == \langle G,(\cdot),(e)\rangle$ (Kuich 1986, p. 5). The definition of a semiring will use the definition of a semigroup.

Definition 2:

A triple $(H,(+),(*))$ is a semiring if and only if:

a) $(H,(+))$ is a commutative semigroup with neutral element (0).

b) $(H,(*))$ is a semigroup with neutral element (1).

c) for all $h_1,h_2,h_3 \in H$,

$$h_1 (*) (h_2 (+) h_3) = (h_1 (*) h_2) (+) (h_1 (*) h_3)$$

$$(h_1 (+) h_2) (*) h_3 = (h_1 (*) h_3) (+) (h_2 (*) h_3)$$

d) for every $h \in H$,

$$(0) (*) h = h (*) (0) = (0)$$

(Kuich 1986, pp. 5 - 6)

For those who understand the concept of a ring, notice that a semiring would be a ring if the additional property of an "inverse" for (+) were added (Hartnett 1963, p. 173). Below are two examples of semirings.

Example 1:

The set of real numbers having ordinary addition with the natural neutral element 0 and multiplication with neutral element 1, $H_1=(\mathbb{R},+,*)$, is a semiring. This can be shown by first proving that $(\mathbb{R},+)$ is a commutative semigroup. Clearly, for all $r_1,r_2 \in \mathbb{R}$ where $\mathbb{R}$ is a nonvoid set, r_1+r_2

$\in \mathbb{R}$, $r_1 + 0 = r_1 + 0 = r_1$. Likewise it can be shown that the real numbers are commutative and associative. The same holds true for $(\mathbb{R},*)$ and it is known that real numbers are distributive and zero absorptive, i.e. part (d) of the definition. (Rote 1985, p. 194)

Example 2:

$H_2=(\mathbb{R} \cup \{-\infty,+\infty\}, \min, +)$ is a semiring with zero $(0)=+\infty$ and unity $(1)=0$, if $(+\infty)+r=(+\infty)$, $(-\infty)+r=(-\infty)$, for all $r \in \mathbb{R}$, and $(-\infty)+(+\infty)=(+\infty)$ are defined. Here it is not as obvious that H_2 is a semiring. However, the min operation as defined normally is commutative, $\min(r_1,r_2)=\min(r_1,r_2)$, and likewise associative. Since $(-\infty)$ and $(+\infty)$ are included with the above definitions + is also commutative and associative. Parts (d) and (e) of the semiring definition also follow to be true.

If $r_1,r_2,r_3 \in (\mathbb{R} \cup \{-\infty,+\infty\})$

(d) $r_1 + \min(r_2,r_3) = r_1+r_2$ OR r_1+r_3 depending on which of r_2 and r_3 are smaller

$= \min((r_1+r_2),(r_1+r_2))$

(e) $(+\infty) + r_1 = r_1 + (+\infty) = (+\infty)$ by the definitions above.

(Rote 1985, p.194)

The definition of the APP as given by Robert and Trystram (1986) follows which includes the definition of a weighted graph.

> Given a weighted graph G=(V,E,w) where V is a finite vertex set, E an arc set, a function w: E -> H with weights from a semiring (H,(+),(*)) with zero (0) and unity (1), find for all pairs of vertices (i,j) the quantities
>
> $$d_{ij} = \bigoplus_{p \in M_{ij}} w(p),$$
>
> where M_{ij} denotes the set of all paths from i to j.

Associated with the weighted graph described above there is a weight matrix $A = (a_{ij})$, where $a_{ij} = w(i,j)$ if $(i,j) \in E$ and $a_{ij} = 0$ otherwise. The set of all paths M_{ij} is modified to $M_{ij}^{(k)}$, the set of all paths from i to j which contain only vertices x with $1 \leq x \leq k$ as intermediate vertices. This will also modify the weight matrix calculations to

$$a_{ij}^{(k)} = \bigoplus_{p \in M_{ij}^{(k)}} w(p)$$

where $d_{ij} = a_{ij}^{(n)}$.

4.0 Corresponding Systolic Array:

A general algorithm was developed by Robert and Trystram (1986) that solves the APP for any particular instance by using only the semiring operations (+), (*), and *, where

$$c^* := (+)c^i = (1) \ (+) \ c \ (+) \ (c(*)c) \ (+) \ (c(*)c(*)c)\cdots$$
$$i>=0$$

The above defintion is the generalized infinite geometric series for 1/(1-c) giving the semiring an "inverse". The algorithm is equivalent to Rote's (1985) algorithm with an improvement in time (Robert 1986, pp. 173, 179). The algorithm is given below:

```
for k <- 1 to n
begin
      a_kk^(k) <- (a_kk^(k-1))*
      for i<- 1 to n, i≠k
            a_ik^(k) <- a_ik^(k-1) (*) a_kk^(k)
      for j <- 1 to n, j≠k
      begin
            for i <- 1 to n, i≠k
                  a_ij^(k) <- a_ij^(k-1) (+) a_ik^(k) (*) a_kj^(k-1)
            a_kj^(k) <- a_kk^(k) (*) a_kj^(k-1)
      end
end
```

The algorithm can solve a variety of applications depending upon the definitions of (+), (*), and * in the semiring. A chart of several applications is given in figure 1. Four additional applications were briefly described in Robert and Trystram (1986) and three are expanded upon here:

1. The determination of the inverse of a real matrix A can be performed by defining (+) and (*) in the usual manner in $\mathbb{R}$ and we have seen that $(\mathbb{R},+,*)$ is indeed a semiring. Since the definition of * is sum of increasing power it can be defined if $c \neq 1$ to be $c^* = 1/(1-c)$. This definition may seem odd since $1/(1-c)$ is not convergent for $abs(c)>1$, but the solution in this case would not exist which is a perfectly adequate solution. The algorithm actually computes $(I-A)^{-1}$ in this case but a simple modification can be done to permit A^{-1} to be computed directly (Rote 1985).
2. The shortest distance in a weighted graph has the definition as follows: $a_{i,j}$ are the weights taken in $H=\mathbb{R} \cup \{-\infty, +\infty\}$, (+) is the minimum operation with zero $(0)=+\infty$, (*) is addition in $\mathbb{R}$ extended to H (with $-\infty$ (*) $+\infty = +\infty$) with $(1) = 0$, and * is defined by if $c>=0$ then $c^* = 0$ else $c^* = -\infty$. Here the semiring is not as obvious however, in example 2 above this was shown to be a semiring.

	Types of problems solved	Problems solved	S	$\oplus$	$*$	i	e
$\oplus$ idempotent	Existence	Problems of connectivity	$\{0, 1\}$	max	min	0	1
	Enumeration	Elementary paths	$\mathscr{P}(X^*)$	union	Latin multiplication	$\varnothing$	X
		Multicriteria problems	$\mathscr{P}(\mathbb{R}^p)$	Set of efficient paths of union	Set of efficient paths of the sum	$(\infty)^p$	$(0)^p$
		Generation of regular languages (Kleene)	Set of words	union	Concatenation	$\varnothing$	The empty word
	Optimization	Path of maximum capacity	$\mathbb{R}^+ \cup \{\infty\}$	max	min	0	∞
		Path with minimum number of arcs	$\mathbb{N} \cup \{\infty\}$	min	+	∞	0
		Shortest path	$\mathbb{R} \cup \{\infty\}$	min	+	∞	0
		Longest path	$\mathbb{R} \cup \{\infty\}$	max	+	$-\infty$	0
		Path of maximum reliability	$\{a \mid 0 \leqslant a \leqslant 1\}$	max	×	0	1
		Reliability of a network	Idempotent polynomials in x	Symmetric difference	×	0	1
$\oplus$ not idempotent	Counting	Counting of paths	$\mathbb{N}$	+	×	0	1
		Markov chains	$\{a \mid 0 \leqslant a \leqslant 1\}$	+	×	0	1
	Optimization and post-optimization	Problems of kth path	Conc of $\mathbb{R}^k$	k smallest elements of two vectors	k smallest terms of sums of pairs	$(\infty)^k$	$(0, \infty, \ldots, \infty)$
		η-optimal paths	Ordered sequence of elements of $\mathbb{R}$ with amplitude η	Sequence of smallest elements up to η of two sequences	Sequence of smallest elements up to η of sums of pairs	∞	0

Figure 1. Applications of the APP

(Gondran 1984)

3. The reflexive and transitive closure of a binary relation can easily be computed with the algorithm. The weight matrix's components $a_{i,j}$ are defined to be boolean, (+) and (*) are respectively the **OR** and **AND** operations, and * is defined by c* = true for all c, see the definition of c*. The weight matrix is the relation matrix for binary relation. Clearly from the concepts of boolean algebra this is a semiring with the zero (0)=0 and unity (1)=1.

Application 3 described above will be the one designed and implemented on a single chip for the CMOS3 process. A practical use for the reflexive and transistive closure is determining if there is a path between any two vertices within a graph. The graph could correspond to cities, nodes in a circuit, etc.

5.0 Systolic Array Function:

The algorithm described above is implemented on a two-dimensional array of n by n+1 orthogonally connected processors (see figure 2). Each row of the array, k, and has n+1 processors labelled P_{k1}, ···, $P_{k,n+1}$. The weight matrix A followed by I_n, which will be represented by C, is fed into the array one row at a time in a staggered fashion (see figure 6).

There are three types of processors which perform different functions. The circle processors implement the *-operation on the first input and afterwards act as delay cells (see figure 3 parts (c) and (d)). The square processors intialize their registers with the modification of the first input and afterward act as multiply-and-add cells (see figure 4 parts (c) and (d)). The double square processors are similiar to the square processors except the register value is initialized differently (see figure 5 parts (c) and (d)). (Robert 1986, p. 174; Robert 1987, p. 187)

Each row k of the array performs the k^{th} phase of the algorithm. Processors P_{k1} transmit the input data arriving from the top to the right. As the data travels to the right, the square processors merely pass this data along uneffected. They modify

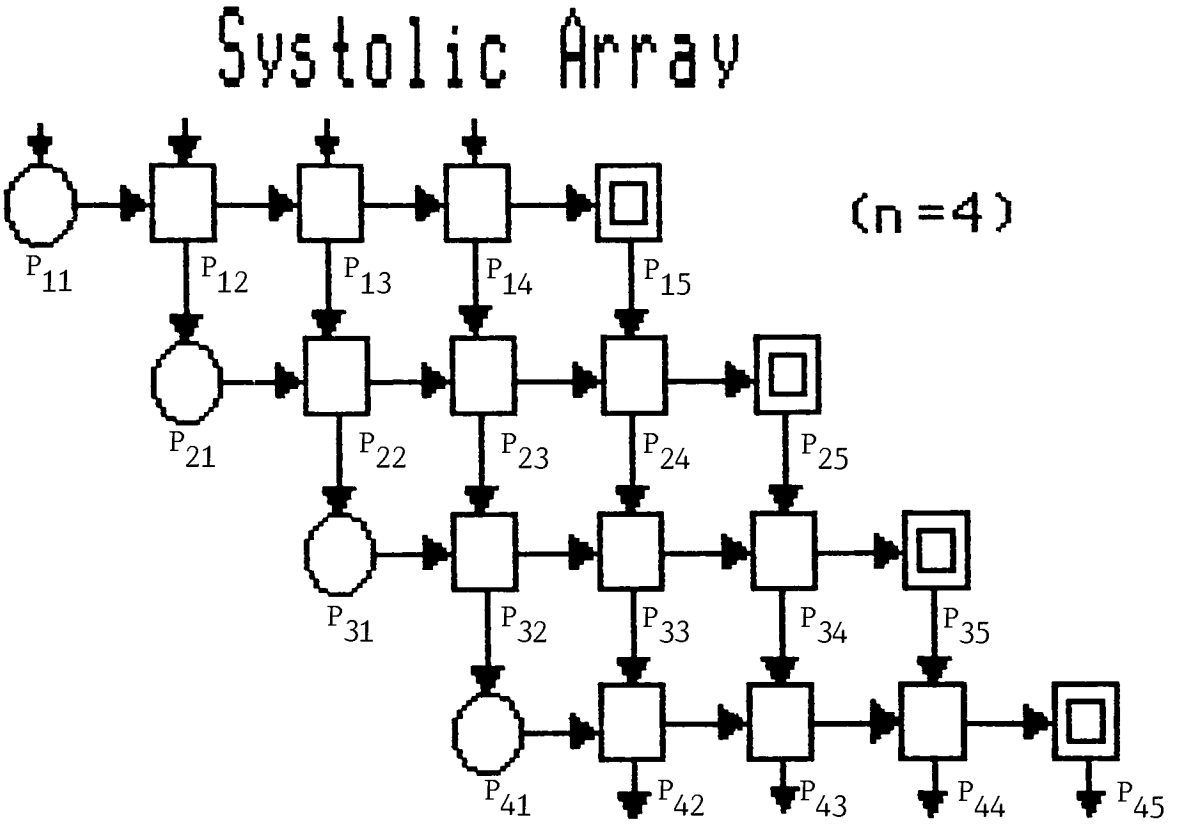
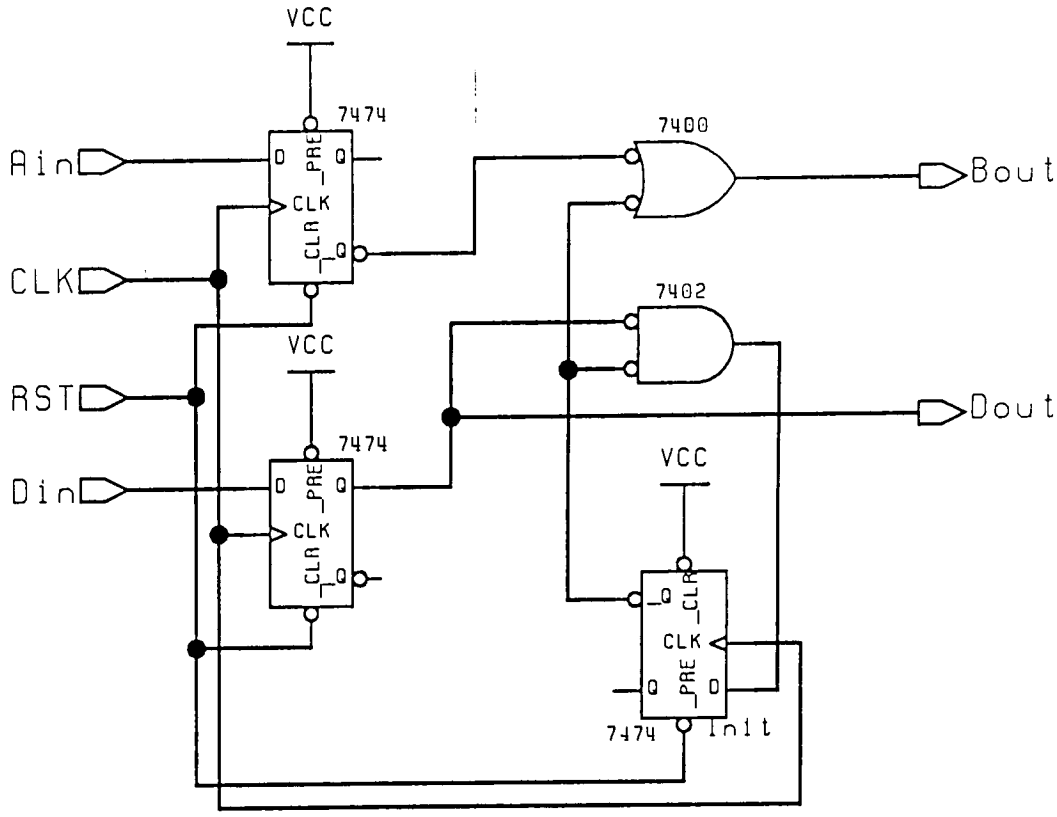


Figure 2. Systolic Array for APP

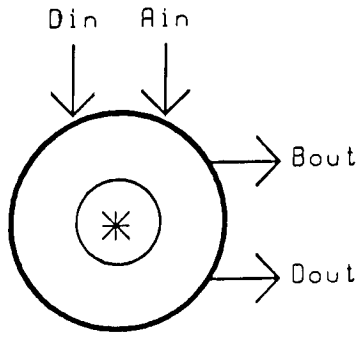
Init	Ain	Bout
0	0	0
0	1	1
1	0	1
1	1	1

(a) truth table



(b) circuit implementation

Init	Din	Init
0	0	0
0	1	0
1	0	1
1	1	0

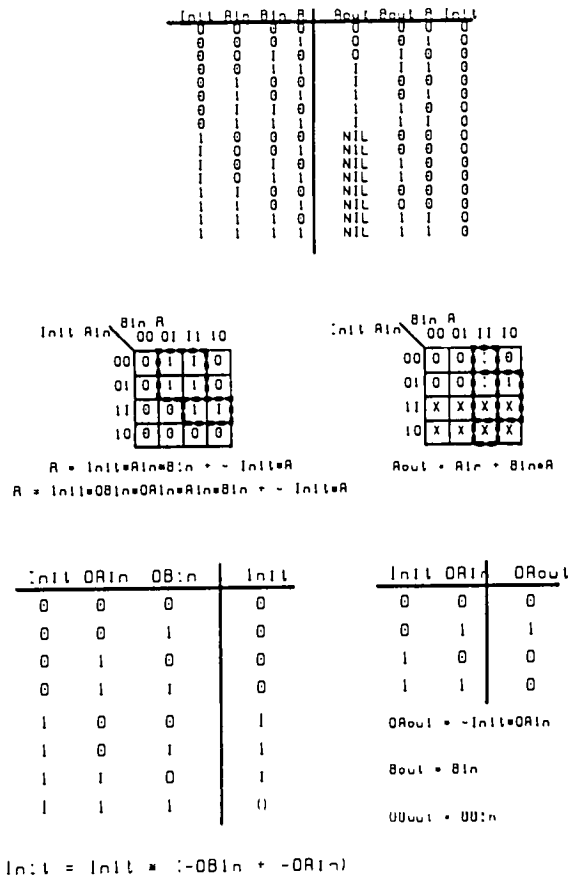


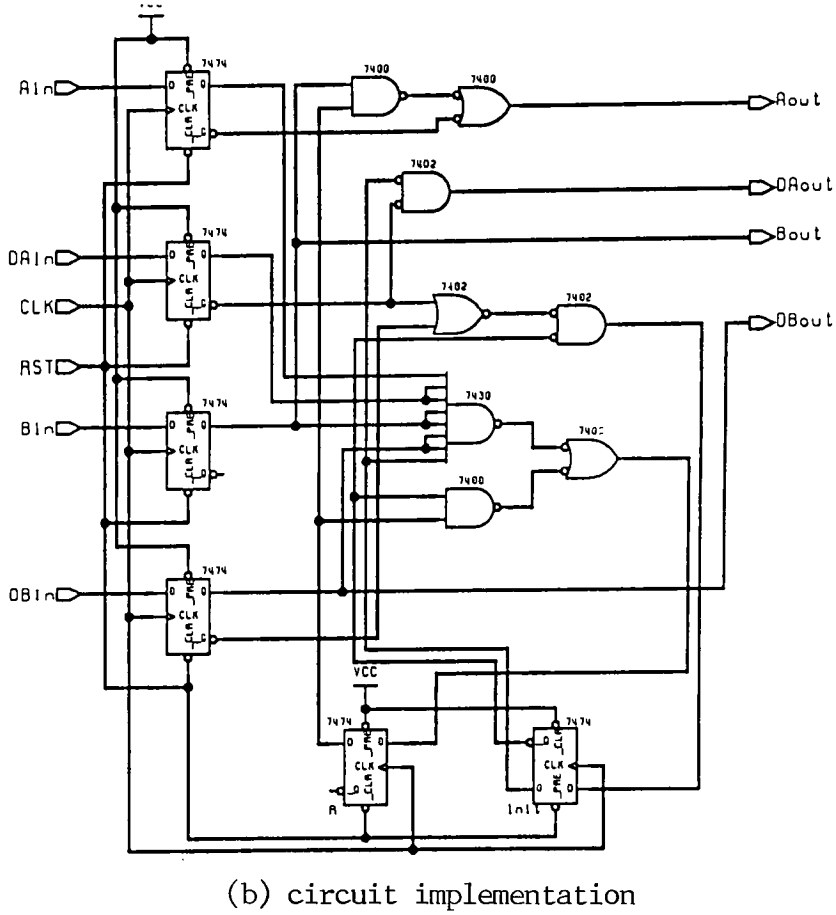
(c) symbol

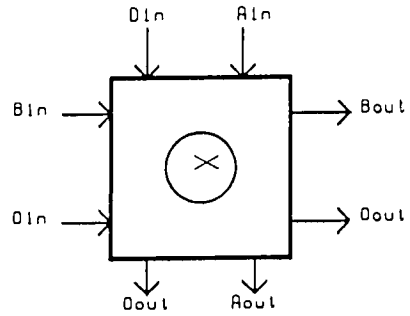
```
IF Init = TRUE THEN
   Bout := Ain*
   Init := FALSE
ELSE
   Bout := Ain
END IF
```

(d) operation

Figure 3.
Circle Processor








```
IF Init = TRUE THEN
   R := Ain and Bin
   Init := FALSE
   Bout := Bin
   Aout := NIL
ELSE
   Aout := Ain or R and Bin
   Bout := Bin
END IF
```

(d) operation

Figure 4.
Square Processor

Init	Bin	Oin	R	Aout	Oout	R	Init
0	0	0	0	X	0	0	0
0	0	0	1	X	0	1	0
0	0	1	0	0	1	0	0
0	0	1	1	1	1	1	0
0	1	0	0	X	0	0	0
0	1	0	1	X	0	1	0
0	1	1	0	1	1	0	0
0	1	1	1	1	1	1	0
1	0	0	0	X	0	X	1
1	0	0	1	X	0	X	1
1	0	1	0	X	0	0	0
1	0	1	1	X	0	0	0
1	1	0	0	X	0	X	1
1	1	0	1	X	0	X	1
1	1	1	0	X	0	1	0
1	1	1	1	X	0	1	0

Init Bin \ Oin R

	00	01	11	10
00	X	X	0	0
01	X	X	1	0
11	X	X	X	X
10	X	X	X	X

Aout = R + Bin

Init Bin \ Oin R

	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	X	X	1	1
10	X	X	0	0

R = Init*Bin + -Init*R

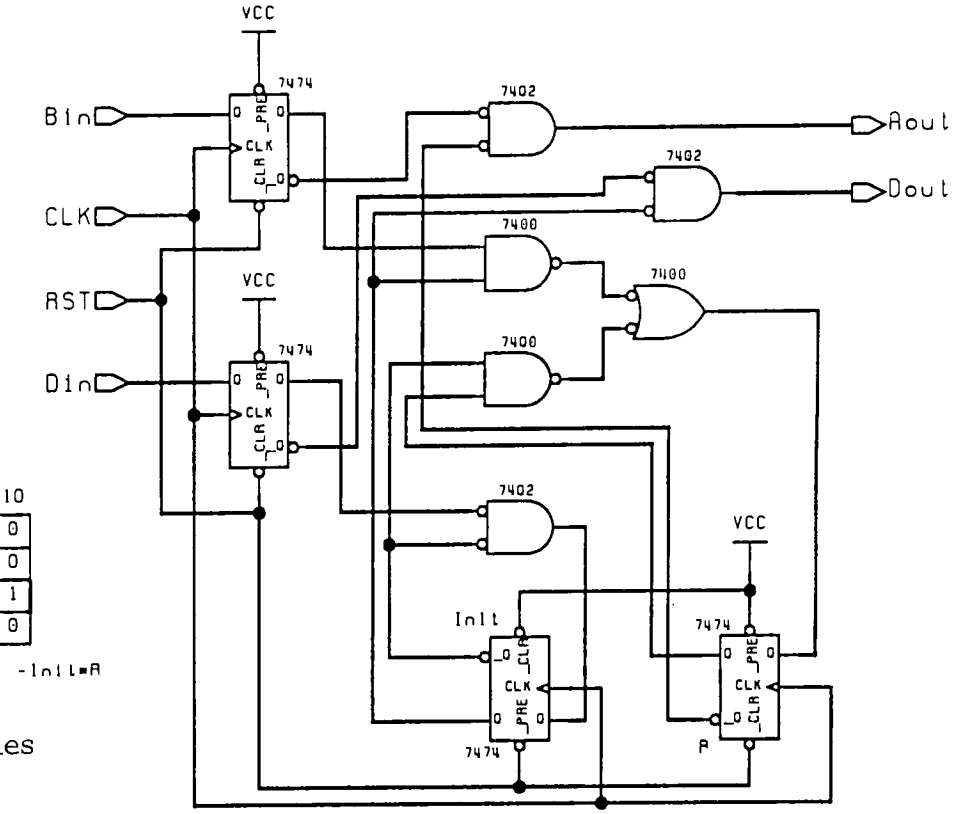
(a) truth tables

Init Bin \ Oin R

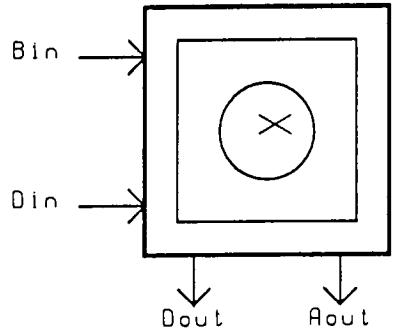
	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	0	0
10	1	1	0	0

Init = Init*-Oin

Oout = -Init * Oin



(b) circuit implementation



(c) symbol

```
IF Init = TRUE THEN
   R := Bin
   Init := FALSE
   Aout := NIL
ELSE
   Aout := R and Bin
END IF
```

(d) operation

Figure 5.
Double Square Processor

and store the first data arriving from the top but modify and pass downward the following data arriving from the top. Likewise the $P_{k,n+1}$ processors store the first input data and modify and pass the following data downward. As seen in figures 3, 4, and 5, the control of the processor's current operation is dependent upon the variable Init.

It can be seen that after 2n input data has gone through the array, its length has been shortened to n, a row of output data. The shortening takes place because each cell that passes data downward does so with its first datum at half the rate in which all data arrives. As the data flows through the array, it is reordered at each row. The element $a_{kk}^{(k)}$ is computed in the circle processor $P_{k,1}$ and moves rightward to be stored in the double-square processor $P_{k,n+1}$. The non-diagonal elements $a_{ik}^{(k)}$, $i \neq k$ are computed by the square processors P_{k2}, $\cdots$, $P_{k,n+1}$ where element $a_{ik}^{(k)}$ is stored in processor $P_{k,(i-k+1) \bmod n}$. Therefore row k of the array outputs matrix $C^{(k)}$ in row order with the leftmost row k+1. (Robert 1986, pp. 174 - 177; Robert 1987, pp. 211 - 213)

The operation of row 1 will be detailed through several steps to aid in the understanding of the data flow.

time step P_{11} P_{12} P_{13} P_{14} P_{15}

1 $a_{11}^{(1)}=(a_{11}^{(0)})^*$

remark: P_{11} receives $a_{11}^{(0)}$ and computes $a_{11}^{(1)}$ and passes it to the right

2 $a_{12}^{(0)}$ $a_{21}^{(1)}$

remark: P_{11} passes $a_{12}^{(0)}$

P_{12} computes $a_{21}^{(1)}=a_{21}^{(0)}(x)a_{11}^{(1)}$ and passes $a_{11}^{(1)}$ to the right.

3 $a_{13}^{(0)}$ $a_{22}^{(1)}$ $a_{31}^{(1)}$

remark: P_{11} passes $a_{13}^{(0)}$,

P_{12} computes $a_{22}^{(1)}=a_{22}^{(0)}(+)a_{21}^{(1)}(x)a_{12}^{(0)}$ and passes it down while passing $a_{12}^{(0)}$ to the right

P_{13} computes $a_{31}^{(1)}=a_{31}^{(0)}(x)a_{11}^{(1)}$ and passes $a_{11}^{(1)}$ to the right

4 $a_{14}^{(0)}$ $*_{23}(1)$ $a_{32}^{(1)}$ $a_{41}^{(1)}$

remark: P_{11} passes $a_{14}^{(0)}$,

P_{12} computes $a_{23}^{(1)}=a_{23}^{(0)}(+)*_{21}(1)(x)*_{13}(0)$ and passes it down while passing $*_{13}(0)$ to the right

P_{13} computes $a_{32}^{(1)}=a_{32}^{(0)}(+)a_{31}^{(1)}(x)a_{12}^{(0)}$ and passes $a_{12}^{(0)}$ to the right

P_{14} computes $a_{41}^{(1)}=a_{41}^{(0)}(x)a_{11}^{(1)}$ and passes $a_{11}^{(1)}$ to the right

5 1 $a_{24}^{(1)}$ $a_{33}^{(1)}$ $a_{42}^{(1)}$ $a_{11}^{(1)}$

remark: P_{11} passes the first 1 of the identity matrix to the right

P_{12} computes $a_{24}^{(1)}$ passing it downward while passing $a_{14}^{(0)}$ to the right

P_{13} computes $a_{33}^{(1)}$ passing it downward and passes $a_{13}^{(0)}$ to the right

P_{14} computes $a_{42}^{(1)}=a_{42}^{(0)}(+)a_{31}^{(1)}(x)a_{12}^{(1)}$ and passes it down while passing $a_{12}^{(0)}$ to the right

P_{15} stores $a_{11}^{(1)}$

6 0 $a_{21}^{(1)}$ $a_{34}^{(1)}$ $a_{43}^{(1)}$ $a_{12}^{(1)}$

remark: P_{11} passes 0 from the identity matrix

P_{12} computes $a_{21}^{(1)}=1(+)a_{21}^{(1)}(x)1$ passing it downward and passing 1 to the right

P_{13} computes $a_{34}^{(1)}$ passing it downward and passes $a_{14}^{(0)}$ to the right

P_{14} computes $a_{43}^{(1)}$ passing it downward and passes $a_{13}^{(0)}$ to the right

P_{15} computes $a_{12}^{(1)}=a_{11}^{(1)}(x)a_{12}^{(0)}$ and passes it downward

The above description is summarized in figures 6 and 7, where the first is the input data and the latter is the output of the the row.

The time neccessary for the array to process the input data is 5n+2 where n is the array size (Robert 1986, p. 174). As previously described, the systolic array size is n(n+1) which will give a size of order n^2. The time and size can be compared to other algorithms and array sizes, see table 1 below.

From the comparisons in table 1, it can be seen that the present algorithm is one of the best in terms of time and array size. Plus, it is a general solution of the APP. The particular instance, the transistive and reflexive closure, that will be implemented with this algorithm has an equal order of time to the one specifically design for it. In fact the last entry in the table, Kung-Lo-Lewis, is a better implementation because it can begin a new solution every n time steps whereas Robert-Trystram's needs 2n steps (Benaini 1989, p. 74).

.	Application	Area	Time
Guibas	transistive closure	n^2	6n
Kung-Lo	shortest paths	n^2	7n
Kramer-Leeuwen	matrix inversion	n^2	6n
Nash-Hansen	matrix inversion	$3n^2/2$	5n
Robert-Tchuente	matrix inversion	n^2	5n
Rote (1985)	general	n^2	7n
Robert-Trystram (1986)	general	n^2	5n
Kung-Lo-Lewis	general	n^2	5n

(Benaini 1989, p.74)

Table 1: Comparisons of systolic arrays for solving APP

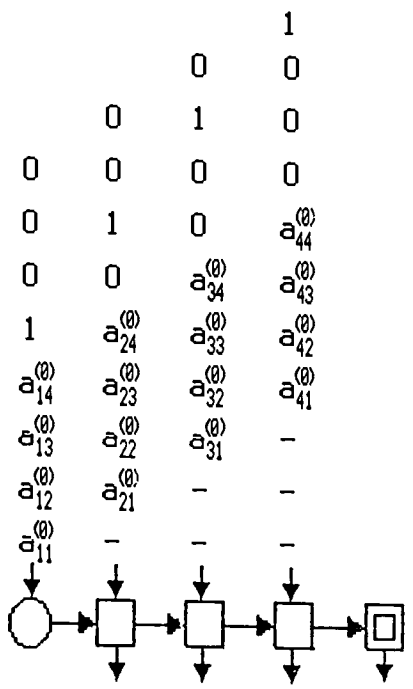


Figure 6.
Input to first row of
the Systolic Array

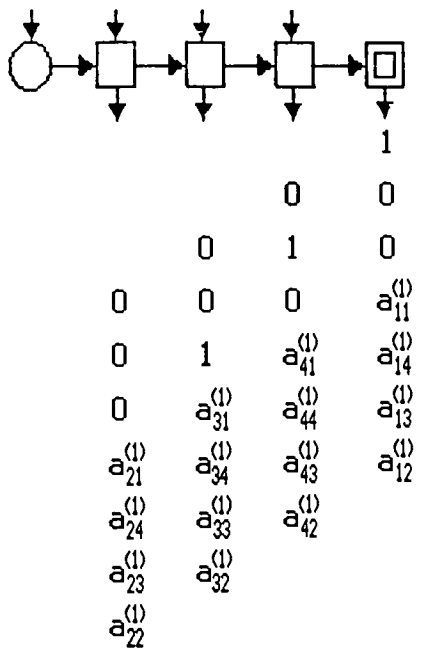


Figure 7.
Output of the first row of
the Systolic Array

6.0 Mapping of Algorithm onto a Systolic Array:

The mapping of Robert and Trystram's algorithm for solving the Algebraic Path Problem (APP) for the specific instance of reflexive and transitive closure of a binary relation is done by

Robert and Trystram's Algorithm to Solve APP

```
 1:   for k <-- 1  to n
 2:   begin
 3:        a_kk^(k) <-- (a_kk^(k-1))*
 4:        for i <-- 1  to n, i≠k
 5:             a_ik^(k) <-- a_ik^(k-1) and a_kk^(k)
 6:        for j <-- 1 to n, j≠k
 7:        begin
 8:             for i <-- 1 to n, i≠k
 9:                  a_ij^(k) <-- a_ij^(k-1) or a_ik^(k) and a_kj^(k-1)
10:             a_kj^(k) <-- a_kk^(k) and a_kj^(k-1)
11:        end
12:  end
```

FIGURE 8

the use of quasi-dependence graphs which look similar to the systolic array depicted in figure 2 except the double square processor is denoted as a single square. The algorithm which is

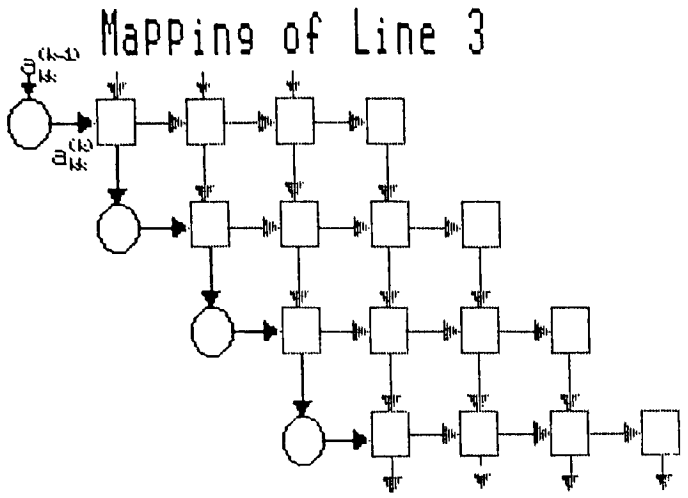


Figure 9.
The mapping of line #3
onto the Systolic Array

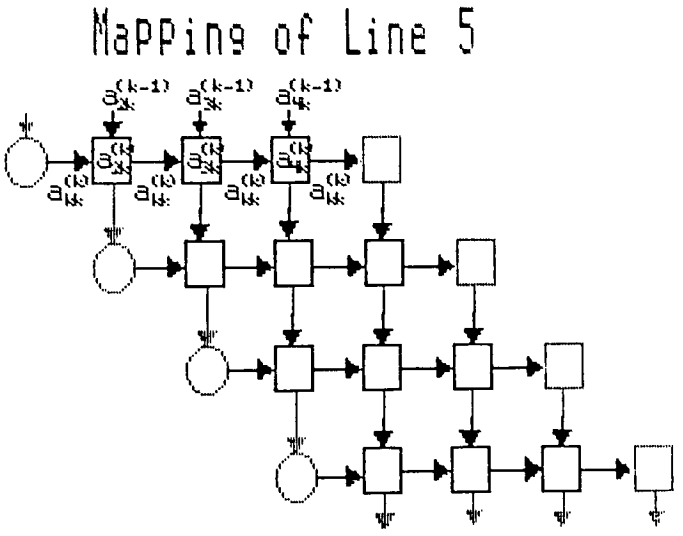


Figure 10.
The mapping of line #5
onto the Systolic Array

listed above in figure 8 has several nested loops. Each relevant instruction line will be discussed separately. In the following description, a relation with four elements will be used. This leads to a 4x4 array. The choice of size 4 is partly because, it is the largest size which can be easily explained, as in the previous section. Also, it limits the number of pins needed on an integrated circuit chip to a reasonable number when the array is fabricated.

6.1 Description of lines:

The first instruction, line #3, has the quasi-dependance graph as illustrated in Figure 9. The value $a_{kk}^{(k)}$ only depends on $a_{kk}^{(k-1)}$ but there needs to be n of these because of the **for** loop in line #1. In Figure 9, the relevant nodes that perform this function are in bold and depicted on the final systolic array to give a feeling of how the mapping will proceede and where the data is flowing.

The next instruction, line #5, has two dependences $a_{ik}^{(k-1)}$ and $a_{kk}^{(k)}$ with one output $a_{ik}^{(k)}$. This relation needs to be repeated for all i between 1 and n except for i=n. The quasi-dependence graph is depicted in Figure 10. Once again there are n repititions because of line #1 and the nodes that perform the

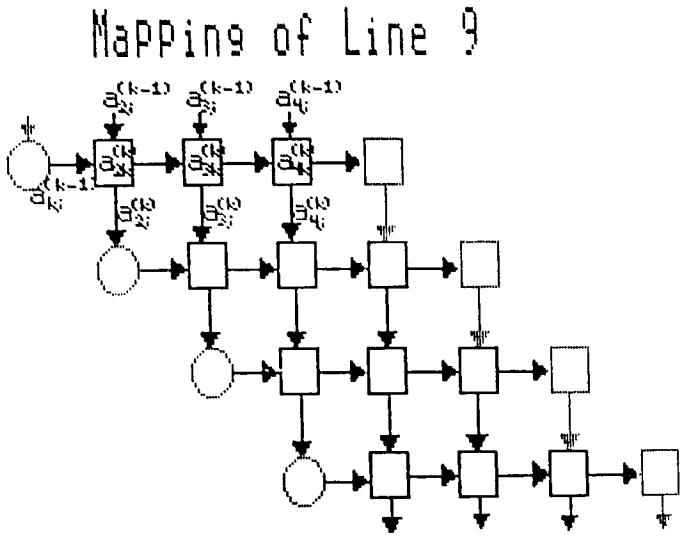


Figure 11.
The mapping of line #9
onto the Systolic Array

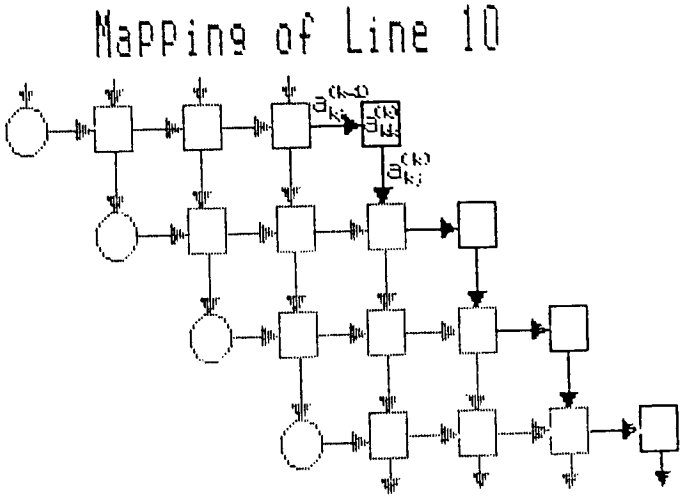


Figure 12.
The mapping of line #10
onto the Systolic Array

operation are bold. Here $a_{ik}^{(k)}$ is shown within the node to indicate that it will be stored there for future use in the final systolic array.

The final loop, line #8, encompasses an instruction that when mapped has three inputs, $a_{ij}^{(k-1)}$, $a_{ik}^{(k)}$, $a_{kj}^{(k-1)}$, and one output, $a_{ij}^{(k)}$. The quasi-dependence graph notation is shown in Figure 11. Here the value that was stored in the previous loop is used as one of the inputs. As seen by the **for** statement, line #8, there are n-1 nodes. The other loops are taken care of by the data flow.

The last instruction, line #10, is depicted in Figure 12. Here one of the two inputs has been stored within the node from previous data flow in the array. This line is in fact taking care of the exclusion of i=k in the previous loop.

7.0 Design and Implementation of Systolic Array:

The choice of the transistive and reflexive closure of a binary relation as the instance for the systolic array to solve was done because of its ease of implementation and the ability to describe its functioning in simple terms. There are other specific algorithm for solving the stated instance, however their time and size are of the same order as this solution O(n) and $O(n^2)$, respectively (see Table 1).

In the following subsections, the necessary equations and circuits are derived and constructed for the systolic array. All the input and output data paths of the processors in figure 2 are actually 2 bits wide as can be seen in figures 3 to 5 part (c). One of the bits is the data, which is labeled either Axx or Bxx for vertically or horizontally flowing, while the other, which is labeled Dxx, signals whether the data is valid. The xx is replaced by in or out for input and output, respectively.

7.1 Logic Equations:

The logic equations for the three different types of cells were derived from the functions each cell needed to perform. In figures 3 through 5 part (d), a programming type implementation

is given at the bottom for the circle, square, and double square processors respectively. The necessary logic equations are derived below in order of processor complexity.

7.1.1 Circle Processor:

The circle processor first performs the *-operation and then acts as a delay cell. These functions are summarized in the truth table at the upper left corner of figure 3(a). When **Init** is true **Bout** should pass **1**, the value of the *-operation, otherwise it should pass **Ain**. The value of **Init** should only remain true if there has not been any data and it is presently true. Valid Data is signalled by **Din** and therefore **Dout** should directly follow **Din**. From these truth tables and the previous description, the following logic equations are derived for the outputs of the circle processor.

Bout = **Ain** or **Init** [1]

Dout = **Din** [2]

Init = -**Din** and **Init** [3]

7.1.2 Double Square Processor:

The double square processor's functioning is depicted at the bottom of figure 5(d). Initially **Bin** is stored within the pro-

cessor in **R** for future computations and no valid value is passed through **Aout**. Afterwards, **Aout** is computed and passed. Since no data is passed the first time **Dout** does not directly follow **Din** however, **Init**'s function remains the same. The truth table summarizing these actions are given in the upper left corner of figure 5(a). The value of **Aout** is only given definite values when **Init** and **Din** are false and true, respectively. The rest are don't cares, X. The reason for this is that **Din** signals when there is valid data. The storage of the first received value **R** should remain the same when **Init** is false but, should latch the value of **Bin** when **Init** and **Din** are both true. Karnaugh maps were used to reduce terms in the equations and are given below the truth table in figure 5(a). The following logic equations were derived:

Aout = **R** and **Bin** [4]

Dout = -**Init** and **Din** [5]

R = (**Init** and **Bin**) or (-**Init** and **R**) [6]

Init = **Init** and -**Din** [7]

7.1.3 Square Processor:

This is the most complex processor since there are two inputs and two outputs. As shown at the bottom of figure 4(d), the processor AND's the initial two inputs, **Ain** and **Bin**, and stores it

in R, while at the same time passing Bin to the right and nothing down. After the first values have been processed, Bin is still passed to the right through Bout and the value passed downward, Aout, is the sum and product expression for Ain, Bin, and R. The Init value depends on two data valid flags, DAin and DBin, and should only remain true when it is true and either of the latter values is false. Since the Aout value is delayed one time step its data valid flag DAout is exactly the same as that for the double square processor. Bin is always passed uneffected and therefore Bout and DBout directly follow their input counterparts Bin and DBin. These conclusions are summarized in the three truth tables on the left side of figure 4(a). Once again Karnaugh maps were used to minimize terms for the complicated equations, R and Aout. There are two equations for R given below its Karnaugh map. The first is the equation directly derived from the mapping while the second has the additional data valid flags added to ensure R only changes when valid data is available. The equations are summarized below:

$$Aout = Ain \text{ or } (Bin \text{ and } R) \quad [8]$$

$$DAout = \neg Init \text{ and } DAin \quad [9]$$

$$Bout = Bin \quad [10]$$

$$DBout = DBin \quad [11]$$

$$R = (Init \text{ and } DBin \text{ and } DAin \text{ and } Ain \text{ and } Bin) \text{ or } (\neg Init \text{ and } R) \quad [12]$$

$$Init = Init \text{ and } (-DBin \text{ or } -DAin) \quad [13]$$

7.2 Circuit Design:

The circuits for the three different processors were designed directly from the derived logic equations. A mixed logic approach was used with NAND and NOR gates. The choice to use exclusively NAND and NOR gates was due to the fact that the design was going to be implemented with cMOS technolgy and these gates are more easily made. The inputs were all latched using D-flip/flops. This was necessary for proper operation, plus it gave both the input and its inverted value. The D-flip/flops were also used to store the internal processor variables, Init and R. A reset line has been included to set all input latches and R to the false state while Init is set to true.

7.2.1 Circle Processor:

The three equations for the circle processor when converted to mixed logic notation become:

[1] $Bout,h = Ain,l \text{ or } Init,l$ [14]

[2] $Dout,h = Din,h$ [15]

[3] $Init,h = -Din,l \text{ and } Init,l$ [16]

In the circuit diagram at the upper right corner of figure 3(b)

these equations are directly implemented.

7.2.2 Double Square Processor:

Similarly, the equations for the double square processor are converted to mixed logic notation for direct implementation using NAND and NOR gates.

[4] **Aout,h** = **R,1** and **Bin,1** [17]

[5] **Dout,h** = -**Init,1** and **Din,1** [18]

[6] **R,h** = (**Init,h** and **Bin,h**),1 or
(-**Init,h** and **R,h**),1 [19]

[7] **Init,h** = **Init,1** and -**Din,1** [20]

Refer to the upper right corner of figure 5(b) for the circuit diagram.

7.2.3 Square Processor:

The final equations for the square processor are converted below. However, in the upper right corner the circuit diagram of figure 4(b), an 8 input NAND gate is used because TTL parts were used during this phase of the design. The 8 input NAND was used as an 5 input NAND by tying together the proper lines.

[8] **Aout,h** = **Ain,1** or (**Bin,h** and **R,h**),1 [21]

[9] **DAout,h** = -**Init,1** and **DAin,1** [22]

[10] $B_{out,h} = B_{in,h}$ [23]

[11] $DB_{out,h} = DB_{in,h}$ [24]

[12] $R,h = (Init,h \text{ and } DB_{in,h} \text{ and } DA_{in,h} \text{ and } A_{in,h} \text{ and } B_{in,h}),1 \text{ or } (\neg Init,h \text{ and } R,h),1$ [25]

[13] $Init,h = Init,1 \text{ and } (\neg DB_{in,h} \text{ or } \neg DA_{in,h}),1$ [26]

8.0 Processor Testing:

Each of the processor cells were simulated by an exhaustive test of the complete truth table for the particular cell. This was done by doing Quicksim simulations on the gate-level circuit designs.

8.1 Circle Processor:

Since there are only two inputs and outputs for this cell a relatively simple set of input forcing functions produced the small truth table given below, table 2. The input forcing function event times in nanoseconds are also given to be compared to the Quicksim simulation output in figure 14.

time	Ain	Din	Bout	Dout	-Init
2000	1	0	1	0	0
3000	1	1	1	1	0
4000	0	0	0	0	1
5000	0	1	0	1	1
6000	1	0	1	0	1
7000	1	1	1	1	1

Table 2: Circle Processor Operation

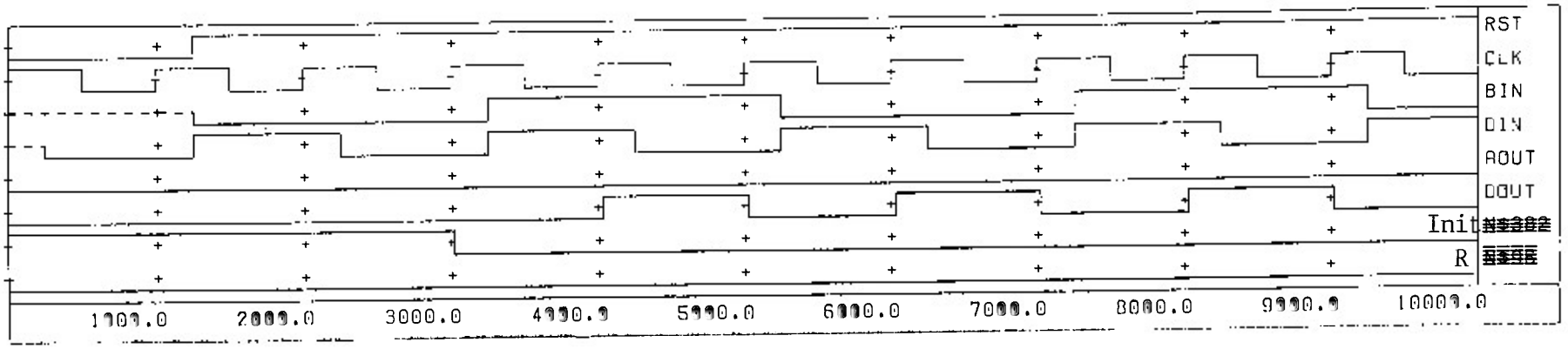


Figure 13. Simulation output for the Double Square Processor (R=0)

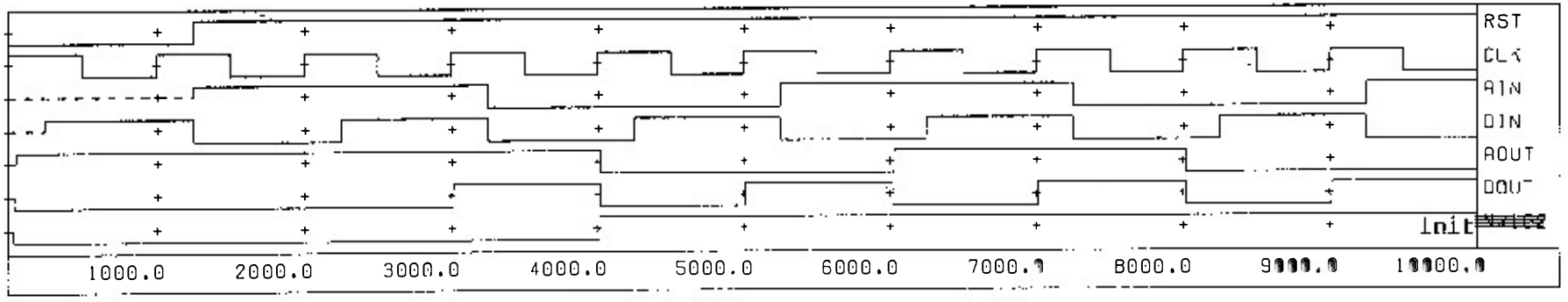


Figure 14. Simulation output for the Circle Processor

The -Init value is the -Q output of the D-flip/flop and should be one time step behind.

This simulation test also proved the proper functioning of the -Init signal because it did only change when there was valid data available. In other tests not shown, the other two variations on inputs **Ain** and **Din** when Init is true were tested and verified. Please refer to figure 3 for the originally created truth tables.

8.2 Double Square Processor:

Similarly to the circle processor, the double square processor has two inputs and outputs. The truth table, table 3, given below has the same format as the one in table 2. The values can be compared to the Quicksim simulation output in figure 13. The Init and R values are the Q outputs of the D-flip/flops and should be one time step behind the other values.

Similar test were done to make R true and are given below, table 4, and in figure 15.

These tests also proved the proper functioning of the Init and R signals. The described tables, 3 and 4, can also be compared

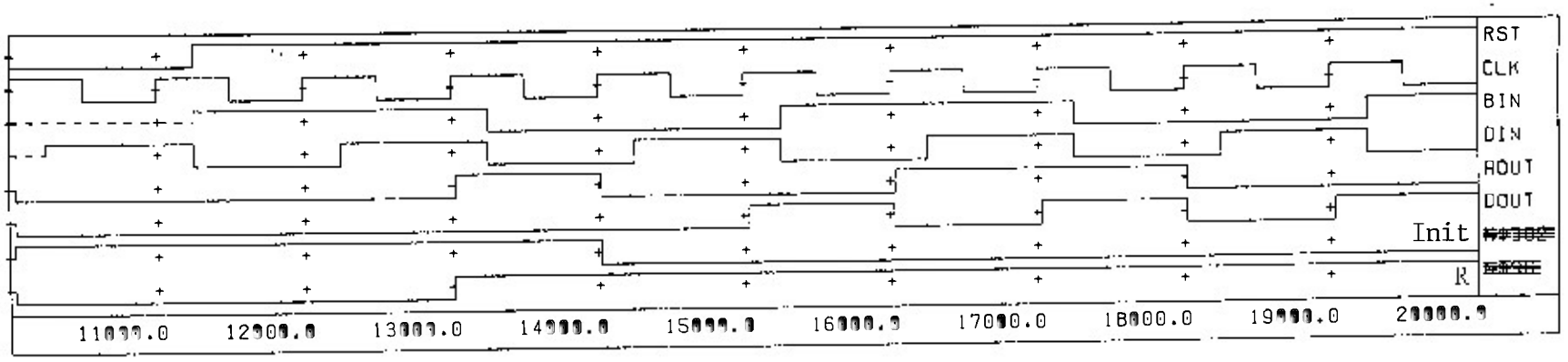


Figure 15. Simulation output for the Double Square Processor (R=1)

to the original truth tables created in figure 5.

time	Bin	Din	Aout	Dout	Init	R
2000	0	1	0	0	1	0
3000	0	0	0	0	0	0
4000	1	1	0	1	0	0
5000	1	0	0	0	0	0
6000	0	1	0	1	0	0
7000	0	0	0	0	0	0

Table 3: Double Square Processor Operation (R=0)

time	Bin	Din	Aout	Dout	Init	R
12000	1	0	0	0	1	0
13000	1	1	1	0	1	1
14000	0	0	0	0	0	1
15000	0	1	0	1	0	1
16000	1	0	1	0	0	1
17000	1	1	1	1	0	1

Table 4: Double Square Processor Operation (R=1)

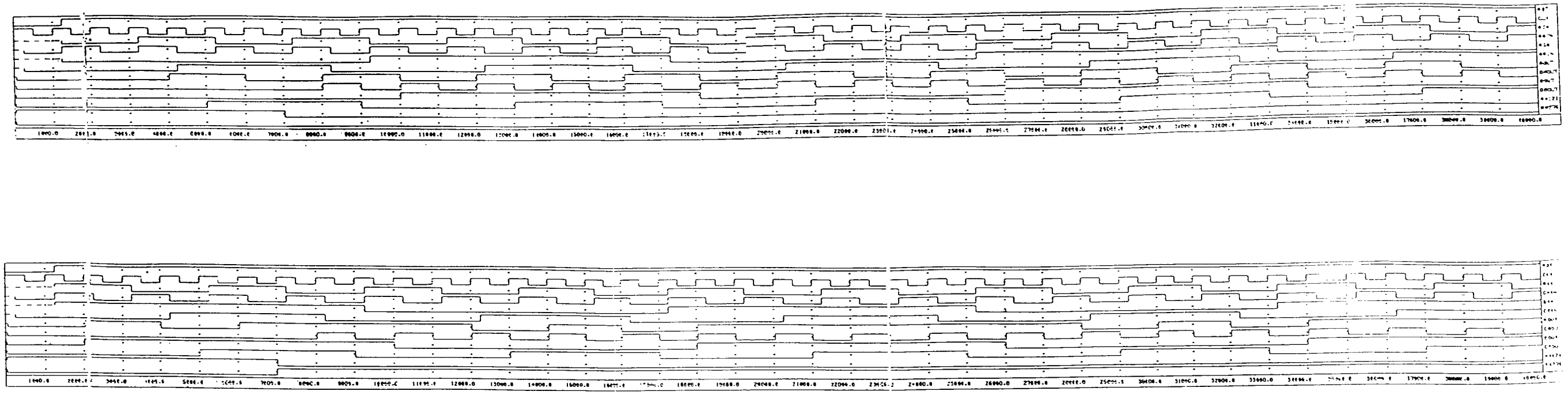


Figure 16. Simulation Output for the Square Processor

8.3 Square Processor:

As with the previous processors, simulation tests were conducted on the square processor. The simulation output is shown in figure 16. A detailed truth table will not be presented here because the listing would be too long and the reader can verify the results using the original truth table in figure 4(a).

9.0 Array Design and Testing:

The processing cells were connected to form several different array sizes for testing. Initially a 2x2 array was created and then a 3x3 and a 4x4 followed. The testing was done using Quicksim for the logic simulations and with a computer program, listed in Appendix A, that strictly followed the program implementation. These two outputs were compared against one another and against hand calculations for correctness. During this phase of the testing it was discovered that there had been an error within one of my main texts (Robert 1986). Luckily the problem was the switching of AND and OR operations for the transistive and reflexive closure, which was thought to be a problem before any implementation began.

The clock period was set at 1000ns to avoid any problems that might occur because of propagation delays or rise and fall times.

9.1 2x2 Array Testing:

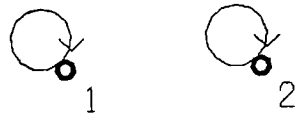
Three different input matrices were used for testing the 2x2 array. The matrices, computer program output, and directed graphs are given in figures 17 to 19. The computer program

Program Input

```
Start: 1  -1
       0   0
       1   1
       0   0
 End:-1   1
```

Given Matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$



Program Output

```
Start: 1  -1
       0   0
 End:-1   1
```

Transistive & Reflexive Closure

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

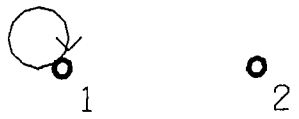
Figure 17. Test #1 for 2x2 Array

Program Input

```
Start: 1 -1
       0  0
       1  0
       0  0
 End:-1  1
```

Given Matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$



Program Output

```
Start: 1 -1
       0  0
 End:-1  1
```

Transistive & Reflexive Closure

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figure 18. Test #2 for 2x2 Array

Program Input

```
Start: 0 -1
       1  0
       1  0
       0  0
 End:-1  1
```

Given Matrix

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

Program Output

```
Start: 1 -1
       0  0
 End:-1  1
```

Transistive & Reflexive Closure

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Figure 19. Test #3 for 2x2 Array

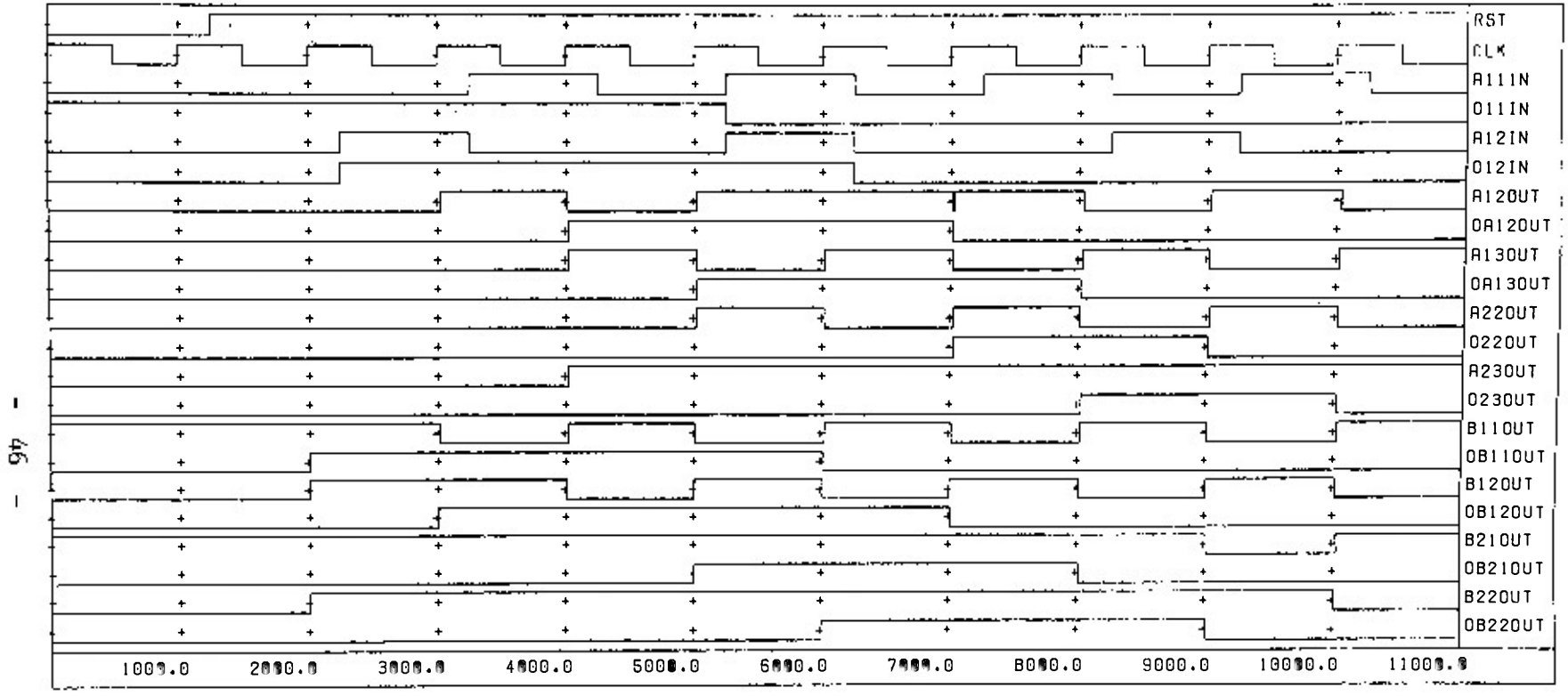


Figure 20. Simulation output for test #1 of 2x2 Array

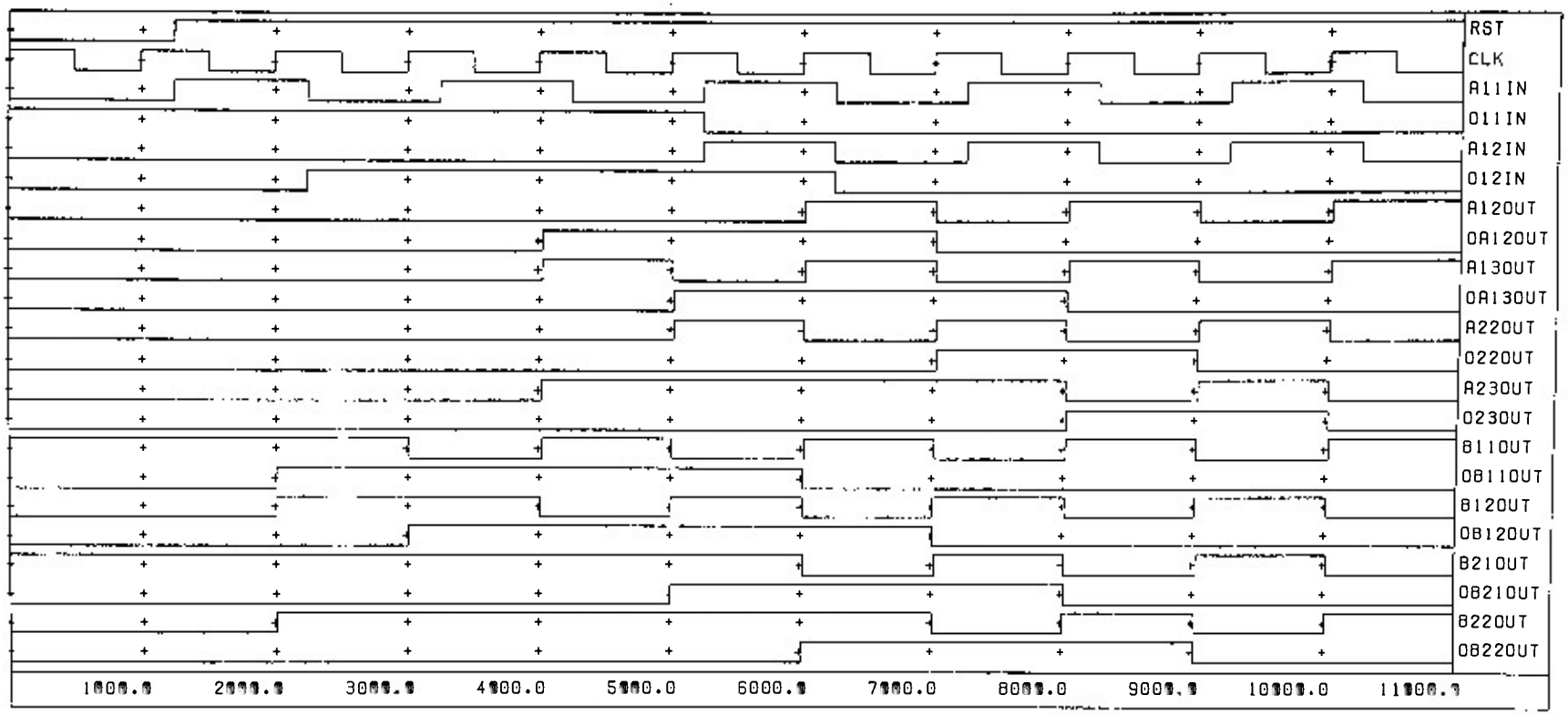


Figure 21. Simulation output for test #2 for 2x2 Array

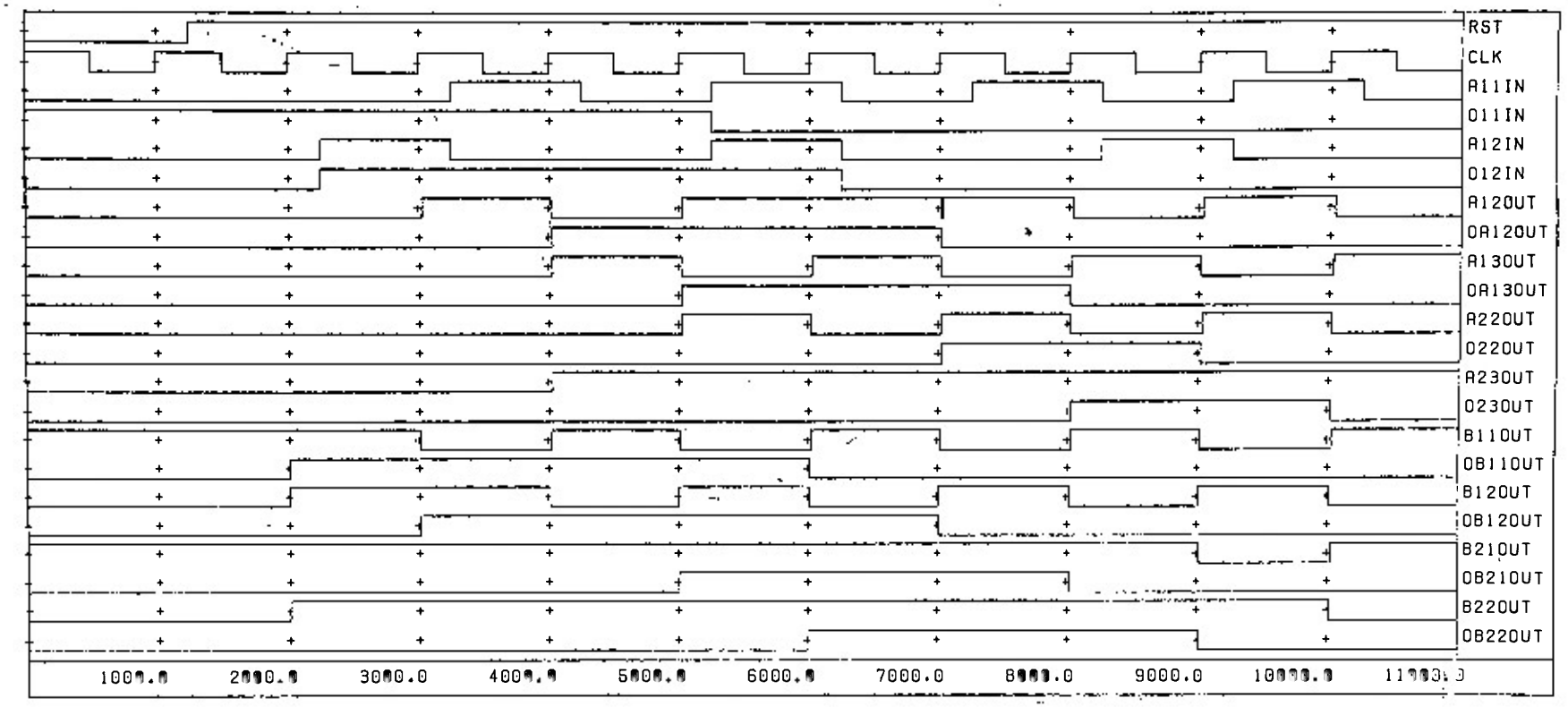


Figure 22. Simulation output for test #3 for 2x2 Array

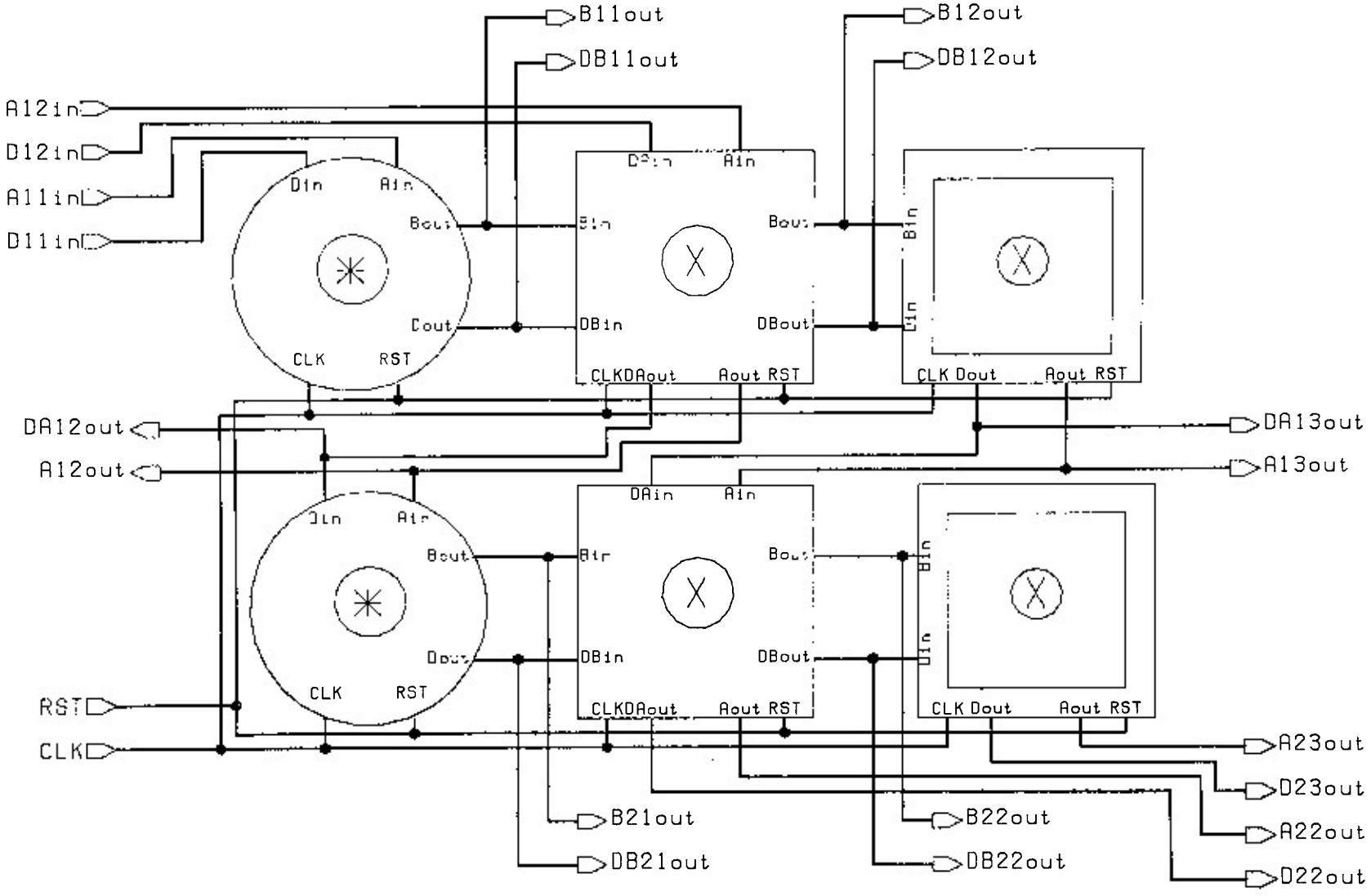


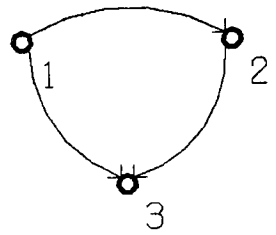
Figure 23. 2x2 Systolic Array

Program Input

```
Start: 0 -1 -1
       1  0 -1
       1  0  0
       1  1  0
       0  0  0
       0  1  0
      -1  0  0
 End:-1 -1  1
```

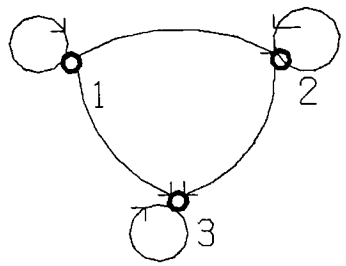
Given Matrix

$$\begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$



Transistive & Reflexive Closure

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$



Program Output

```
Start: 1 -1 -1
       1  0 -1
       1  1  0
      -1  1  0
 End:-1 -1  1
```

Figure 24. Test #1 for 3x3 Array

Program Input

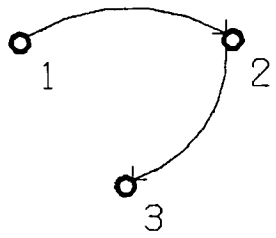
```
Start: 0 -1 -1
       1  0 -1
       0  0  0
       1  1  0
       0  0  0
       0  1  0
      -1  0  0
 End: -1 -1  1
```

Program Output

```
Start: 1 -1 -1
       1  0 -1
       1  1  0
      -1  1  0
 End: -1 -1  1
```

Given Matrix

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$



Transistive & Reflexive Closure

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

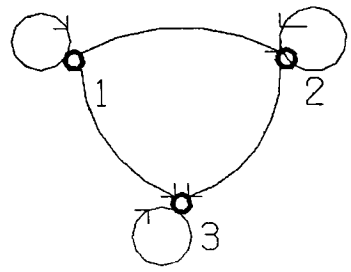


Figure 25. Test #2 for 3x3 Array

showed the data flow into and out of the array and simulated invalid data as a -1. This can be compare with figures 6 and 7 to see the skewed data input and output.

The Quicksim simulation outputs are respectively given in figures 20 to 22. These simulations show the values of all inputs, outputs, and intermediate points. The matrices in figures 17 to 19 were compared to the simulations.

The symbolic diagram of the array is depicted in figure 23. At this stage all lines were tested for correct operation. The computer program was able to print the state of the array at each time interval and was used to compare against the Quicksim output for correctness. Thorough testing on this 2x2 array prevented major problems with any larger sizes.

9.2 3x3 Array Testing:

Two input matrices were tested against a 3x3 array, see figures 24 to 25. The Quicksim simulations are not included for sake of brevity, however they were done and compared with the computer simulations and hand calculations.

9.3 4x4 Array Testing:

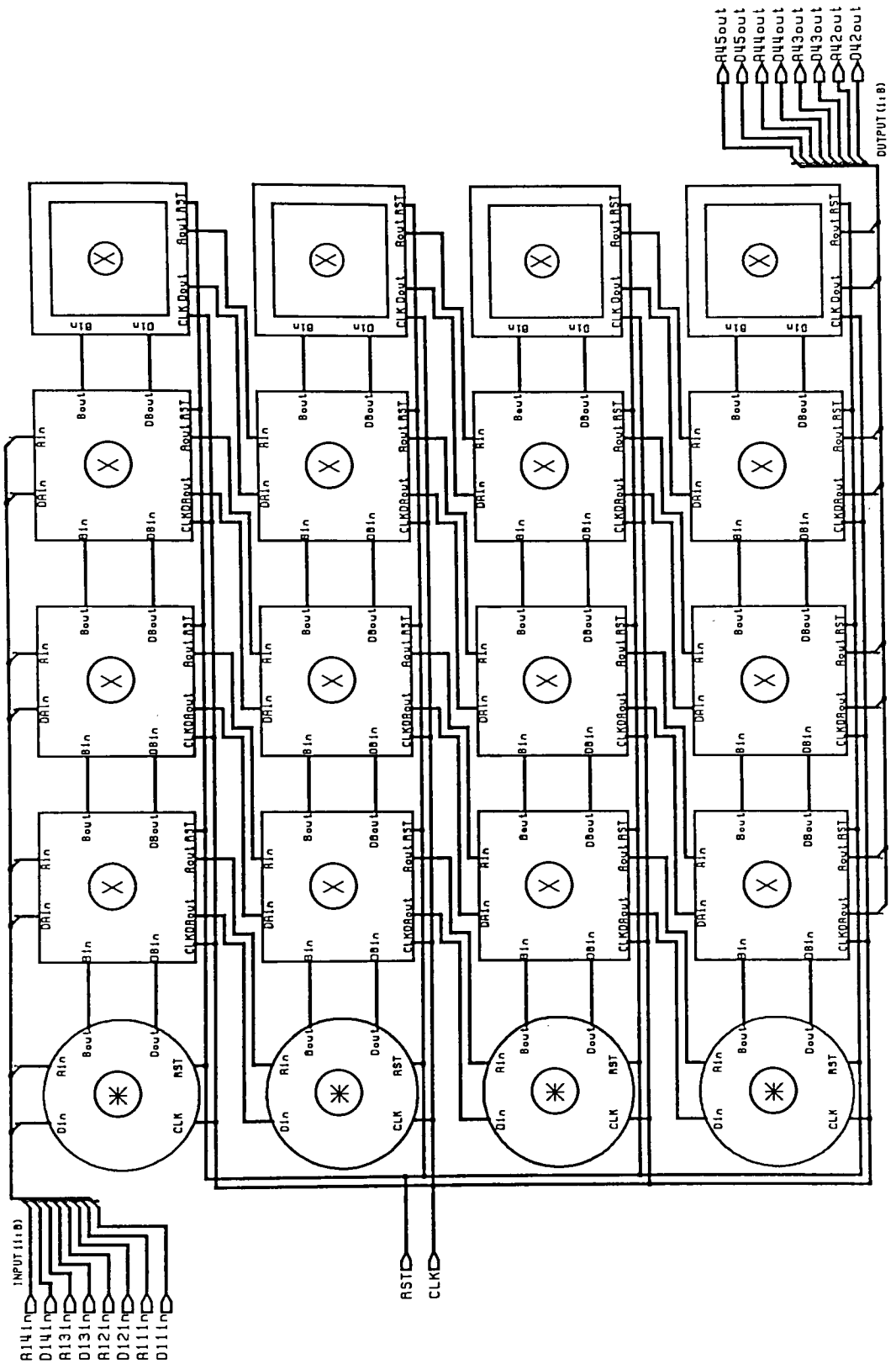


Figure 26. 4x4 Systolic Array

Program Input

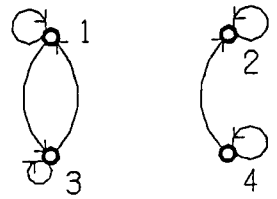
```
Start: 1  -1  -1  -1
       0   0  -1  -1
       1   1   1  -1
       0   0   0   0
       1   0   1   1
       0   0   0   0
       0   1   0   1
       0   0   0   0
      -1   0   1   0
      -1  -1   0   0
End:  -1  -1  -1   1
```

Program Output

```
Start: 1  -1  -1  -1
       0   0  -1  -1
       1   1   1  -1
       0   0   0   0
      -1   0   1   1
      -1  -1   0   0
End:  -1  -1  -1   1
```

Given Matrix

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$



Transistive & Reflexive Closure

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

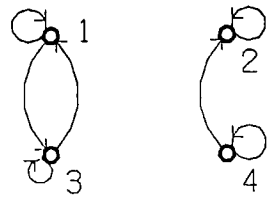


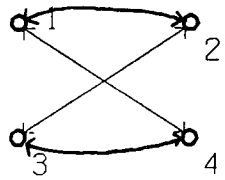
Figure 27. Test #1 for 4x4 Array

Program Input

```
Start: 0 -1 -1 -1
       1  1 -1 -1
       0  0  0 -1
       1  1  1  1
       1  1  0  1
       0  0  1  1
       0  1  0  0
       0  0  0  0
      -1  0  1  0
      -1 -1  0  0
  End:-1 -1 -1  1
```

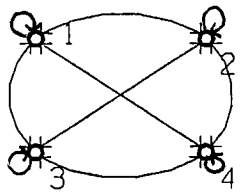
Given Matrix

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$



Transistive & Reflexive Closure

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$



Program Output

```
Start: 1 -1 -1 -1
       1  1 -1 -1
       1  1  1 -1
       1  1  1  1
      -1  1  1  1
      -1 -1  1  1
  End:-1 -1 -1  1
```

Figure 28. Test #2 for 4x4 Array

Program Input

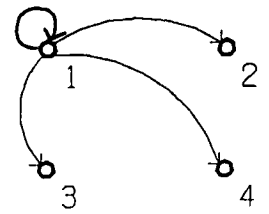
```
Start: 1 -1 -1 -1
       1  0 -1 -1
       1  0  0 -1
       1  0  0  0
       1  0  0  0
       0  0  0  0
       0  1  0  0
       0  0  0  0
      -1  0  1  0
      -1 -1  0  0
 End: -1 -1 -1  1
```

Program Output

```
Start: 1 -1 -1 -1
       1  0 -1 -1
       1  1  0 -1
       1  0  0  0
      -1  0  1  0
      -1 -1  0  0
 End: -1 -1 -1  1
```

Given Matrix

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$



Transistive & Reflexive Closure

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

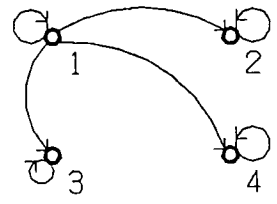


Figure 29. Test #3 for 4x4 Array

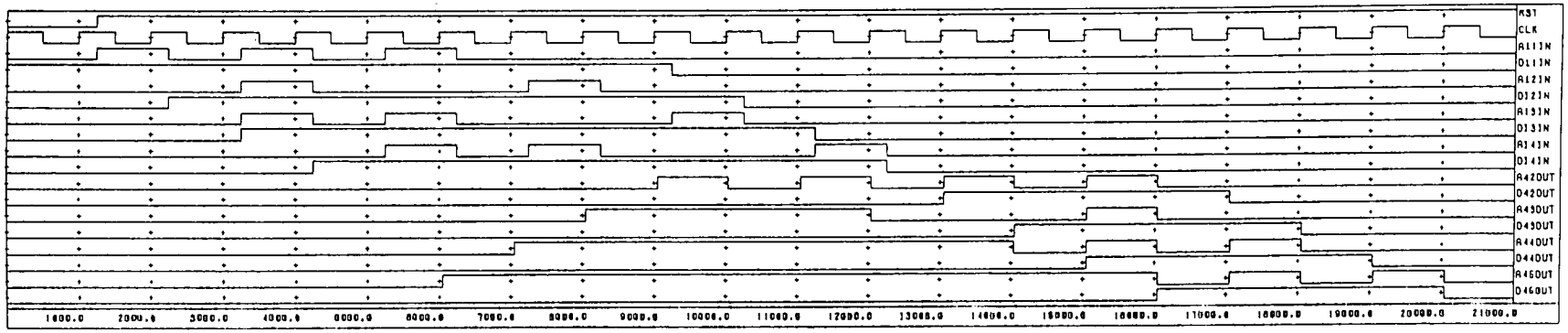


Figure 30. Simulation output for test #1 on 4x4 Array

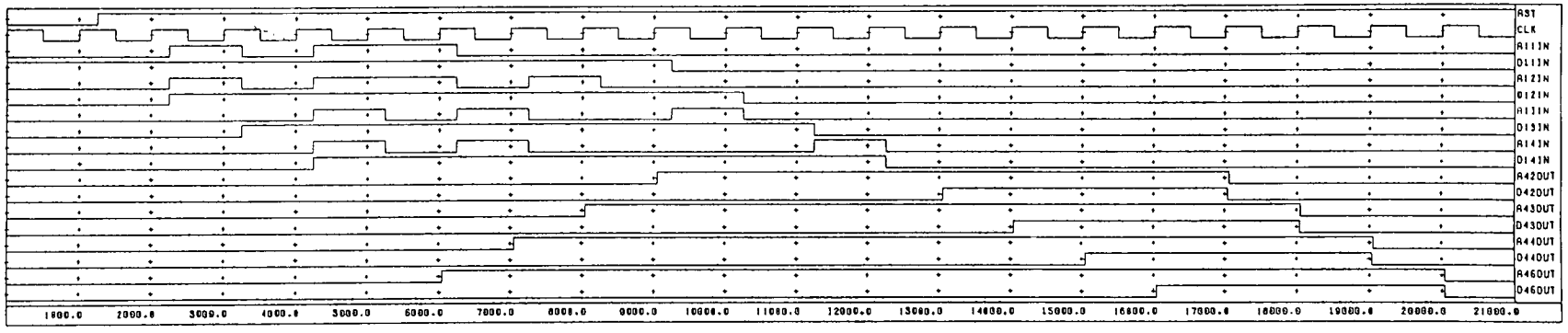


Figure 31. Simulation output for test #2 on 4x4 Array

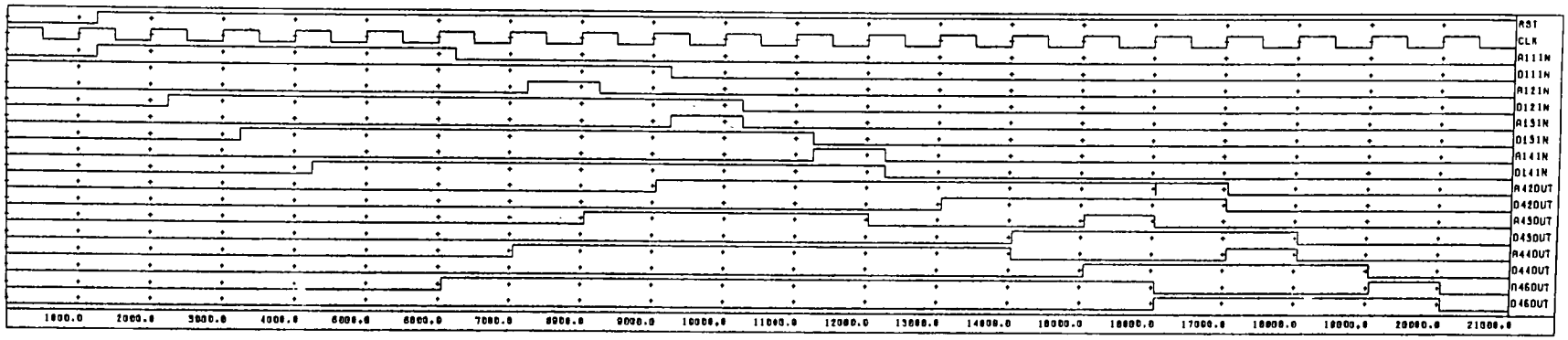


Figure 32. Simulation output for test #3 on 4x4 Array

The circuit diagram for a 4x4 array is depicted in figure 26. It is merely the logical expansion of the 2x2 array. During the array testing, the intermediate points were not displayed in the Quicksim simulations because they were checked during the 2x2 testing and the information would be too overwhelming for easy and accurate verification. The three input matrices tested are depicted in figrures 27 to 29 along with the Quicksim simulation in figures 30 to 32, respectively.

10.0 Conversion of Design to MOSIS3 Standard Cells:

The 4x4 array can be made into an integrated circuit using MOSIS CMOS3 (MOSIS3) standard cells. Mentor Graphic's Corporation's Cell Station design automation tools running on the Apollo workstations were used to layout the design. The chip can be fabricated with these standard cells using the MOSIS facility. The TTL component design therefore must be converted to standard cell formats, see figure 33.

The NAND and NOR gates were directly converted except that the mixed logic symbols were not available, see figures 34 to 36. In the square processor, the 8-input NAND gate which was being used as an 5-input NAND gate had to be converted to a two stage implementation using two NANDs and one NOR, see figures 3 and 35.

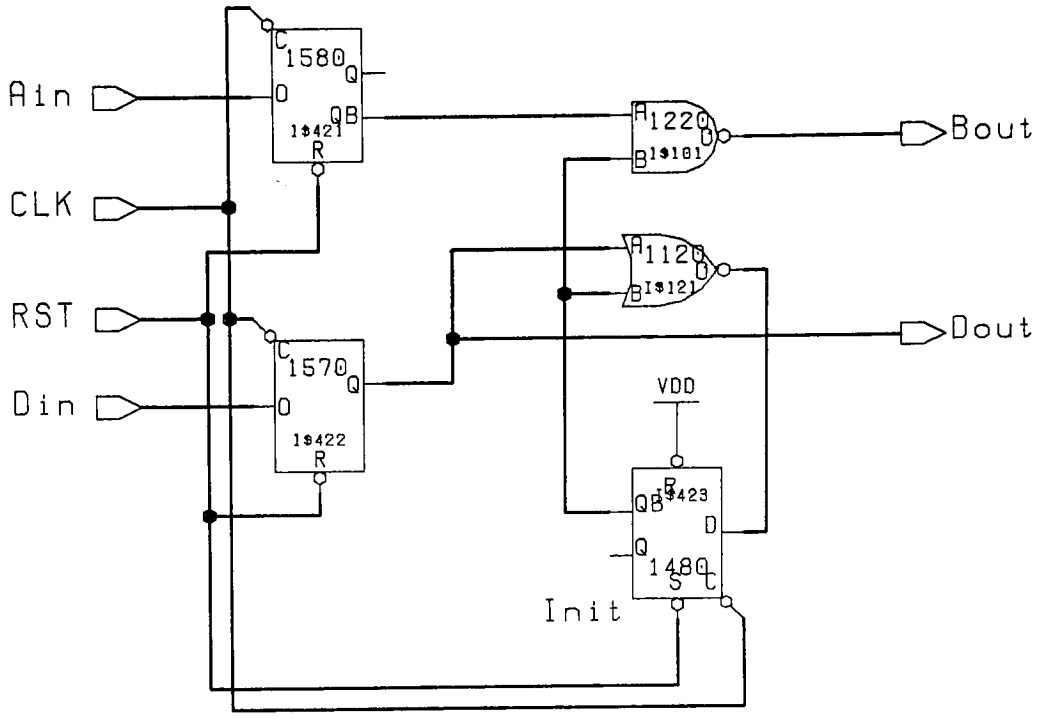
Most of the D-flip/flops were changed to ones that had only the neccessary outputs and inputs, i.e. only Q and/or only R. Latches were going to be used instead of the D-flip/flops, but after checking propgation delays and other timing requiremnents it was discovered that the flip/flops were as fast or faster than the corrsponding latch with only a small increase in area size (Heinbuch 1988).

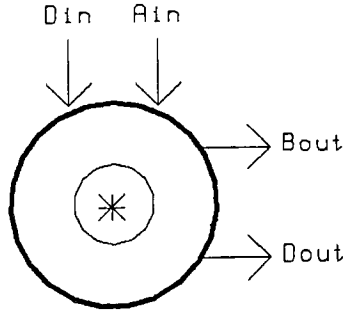
```
Command                                  Description
1                    MOSIS_NETED file [ NETED version for MOSIS3 ]
2              MOSIS_EXPAND_COMP file CMOS3 [ Builds netlist ]
3           MOSIS_DESIGN_CHECKER file CMOS3 [ Electrical Check ]
4            MOSIS_EXPAND_DESIGN file CMOS3 [ Flattens Design ]
5                        QUICKSIM file [ Simulates Circuit ]
6                MOSIS_ADD_DELAY file CMOS3 -FO [ Adds Fanout Delay ]
7                        QUICKSIM file [ Simulates Circuit ]
8   MOSIS_LAYOUT LOGIC_ENTRY file CMOS3 [Creates physical design]
9     MOSIS_LAYOUT CELLFLOOR file CMOS3 [ Generates floorplan ]
10   MOSIS_LAYOUT EDIT_PARMS file CMOS3 [ Edit floorplan ]
11    MOSIS_LAYOUT CELLFLOOR file CMOS3 [ Must rerun after edit ]
12    MOSIS_LAYOUT CELLPLACE file CMOS3 [ Placement of cells ]
13    MOSIS_LAYOUT CELLPOWER file CMOS3 [ Routes power networks ]
14    MOSIS_LAYOUT CELLROUTE file CMOS3 [ Routes signal networks ]
15 MOSIS_LAYOUT CELLSQUEEZE file CMOS3 [ Removes excess in
                                          routing channels ]
16     MOSIS_LAYOUT MINROUTE file CMOS3 [ Minimizes use of poly ]
17             MOSIS_ADD_DELAY file CMOS3 -BA -LO [ Adds back
                                                    annotation and
                                                    wiring delays ]
18                       QUICKSIM file [ Simulates Circuit ]
19     MOSIS_LAYOUT PREGRAPH file CMOS3 [ Generates working file ]
20    MOSIS_LAYOUT CELLGRAPH file CMOS3 [ Allows manual editing ]
21   MOSIS_LAYOUT CELLVERIFY file CMOS3 [ Final validity check ]
22   MOSIS_LAYOUT GDS2_OUPUT file CMOS3 [ Generates layout
                                          information in proper
                                          format for fabrication ]
```

Figure 33. Steps to convert to MOSIS3 standard cells and produce fabrication file

Init	Ain	Bout
0	0	0
0	1	1
1	0	1
1	1	1

Init	Din	Init
0	0	0
0	1	0
1	0	1
1	1	0






```
IF Init = TRUE THEN
    Bout := Ain*
    Init := FALSE
ELSE
    Bout := Ain
END IF
```

Figure 34.
Circle Processor
using MOSIS3
Standard Cells

Init	Ain	Bin	R	Aout	Bout	R	Init
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0
0	0	1	0	0	1	0	0
0	0	1	1	1	1	0	0
0	1	0	0	1	0	0	0
0	1	0	1	1	0	1	0
0	1	1	0	1	1	0	0
0	1	1	1	1	1	1	0
1	0	0	0	NIL	0	0	0
1	0	0	1	NIL	0	0	0
1	0	1	0	NIL	1	0	0
1	0	1	1	NIL	1	0	0
1	1	0	0	NIL	0	0	0
1	1	0	1	NIL	0	0	0
1	1	1	0	NIL	1	1	0
1	1	1	1	NIL	1	1	0

Init Ain \ Bin R	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	0	1	1
10	0	0	0	0

R = Init*Ain*Bin + ~Init*R

R = Init*OBin*OAin*Ain*Bin + ~Init*R

Init Ain \ Bin R	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	X	X	X	X
10	X	X	X	X

Aout = Ain + Bin*R

Init	DAin	DBin	Init
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Init = Init * (~DBin + ~DAin)

Init	OAin	OAout
0	0	0
0	1	1
1	0	0
1	1	0

OAout = ~Init*OAin

Bout = Bin

OBout = DBin

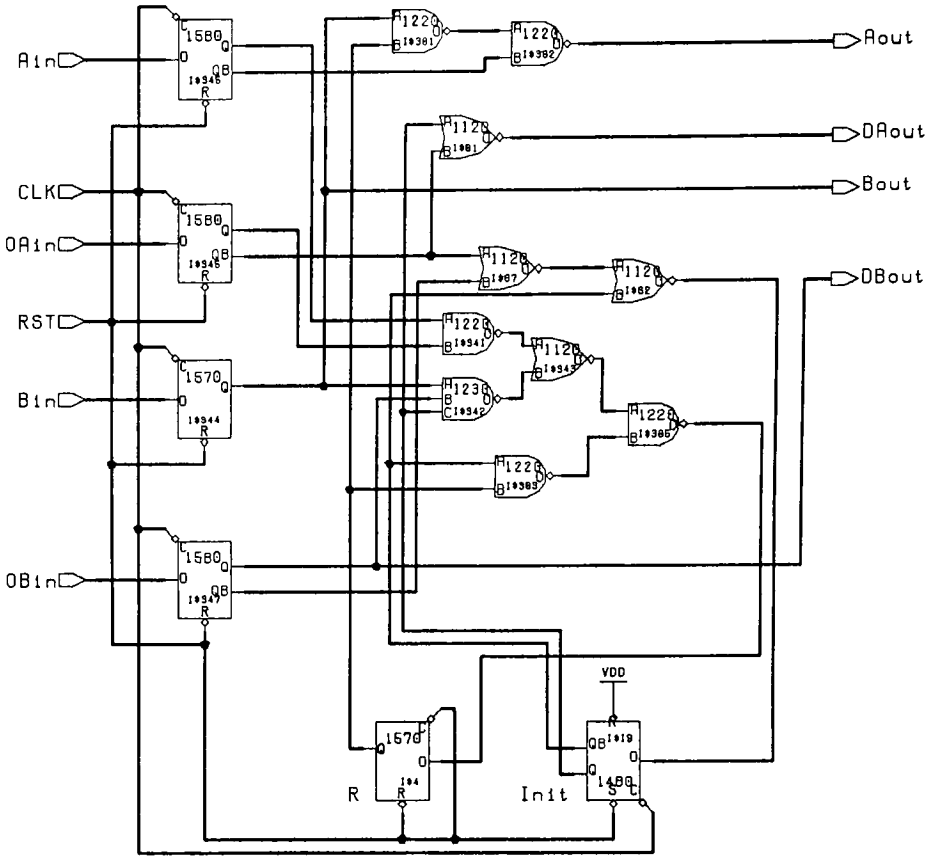
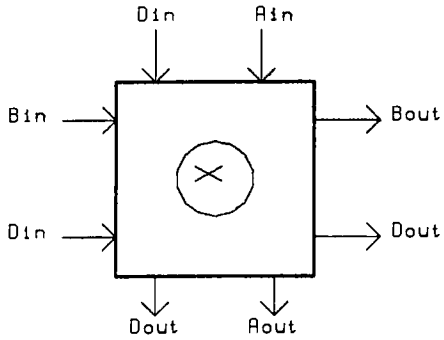
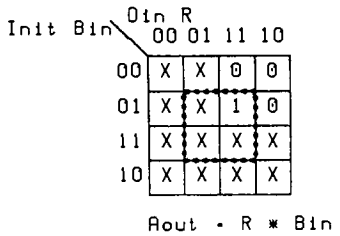


Figure 35.
Square Processor
using MOSIS3
Standard Cells

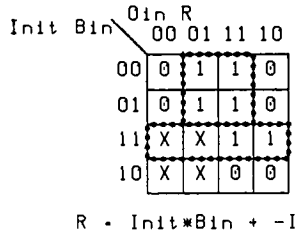



```
IF Init = TRUE THEN
   R := Ain and Bin
   Init := FALSE
   Bout := Bin
   Aout := NIL
ELSE
   Aout := Ain or R and Bin
   Bout := Bin
END IF
```

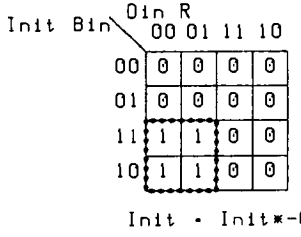
Init	Bin	Oin	R	Aout	Oout	R	Init
0	0	0	0	X	0	0	0
0	0	0	1	X	0	1	0
0	0	1	0	0	1	0	0
0	0	1	1	0	1	1	0
0	1	0	0	X	0	0	0
0	1	0	1	X	0	1	0
0	1	1	0	0	1	0	0
0	1	1	1	1	1	1	0
1	0	0	0	X	0	X	1
1	0	0	1	X	0	X	1
1	0	1	0	X	0	0	0
1	0	1	1	X	0	0	0
1	1	0	0	X	0	X	1
1	1	0	1	X	0	X	1
1	1	1	0	X	0	1	0
1	1	1	1	X	0	1	0



Aout = R * Bin

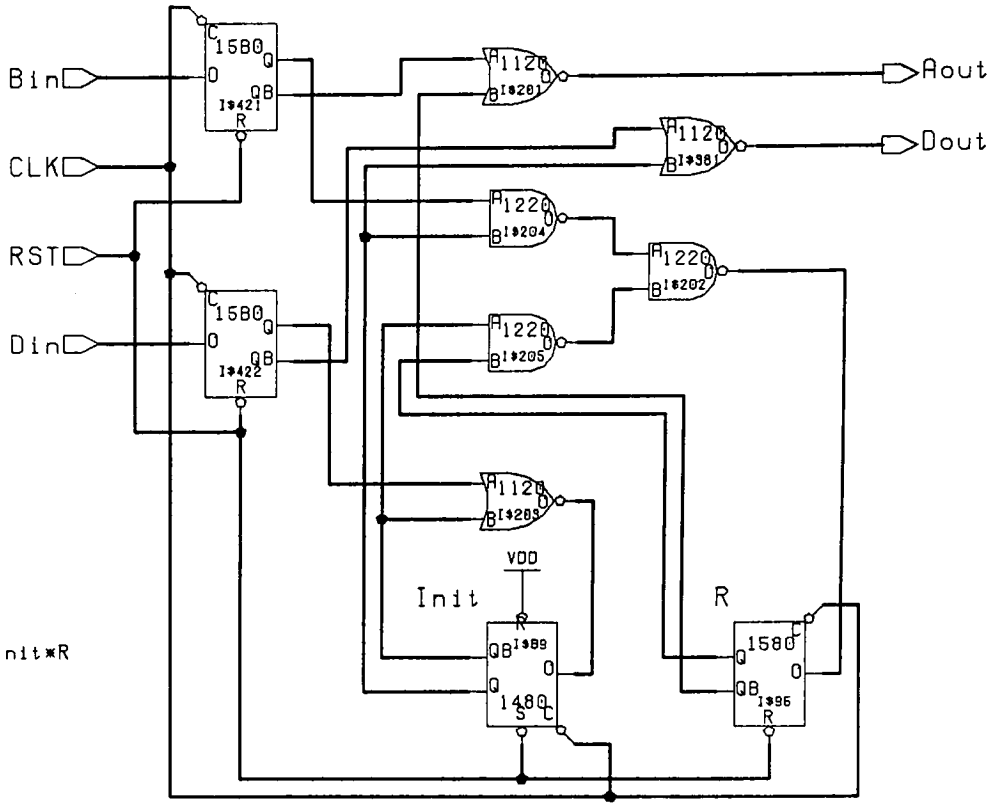


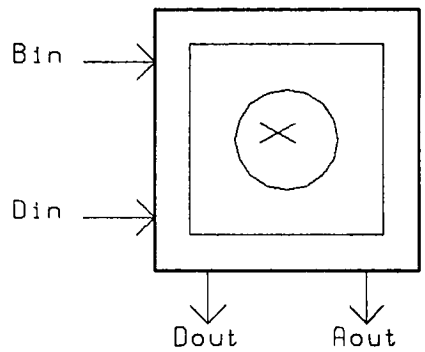
R = Init*Bin + -Init*R



Init = Init*-Oin

Oout = -Init * Oin






```
IF Init = TRUE THEN
   R := Bin
   Init := FALSE
   Aout := NIL
ELSE
   Aout := R and Bin
END IF
```

Figure 36.
Double Square Processor
using MOSIS3
Standard Cells

The I/O inputs for the TTL implementation had to be changed to I/O pads for the chip layout. Additionally, Vdd and Vss pads needed to be included within the circuit diagram for proper layout. The circuit diagram is given in figure 37 and was developed using Mentor Graphic's Schematic Editor, NETED, with CMOS3 circuit elements. The editor can be called inot operation using the command MOSIS_NETED. A complete listing of the additional commands and steps necessary for the conversion to a MOSIS3 standard cell layout is given in figure 33.

The 4x4 array was chosen for the chip layout because the 20 pins required will fit within the standard 24 pin chip. The next array size which would be useful, an 8x8, would require 70 pins.

The array's outputs were not latched. This may give glitches on these lines during the zero state of the clock. The reasons for not latching the outputs are: (1) More than one chip could be connected together then, the latching would be redundant; (2) The latching can be done externally, especially for the additional design which would be discussed later where several chips could be connected together to form any array size.

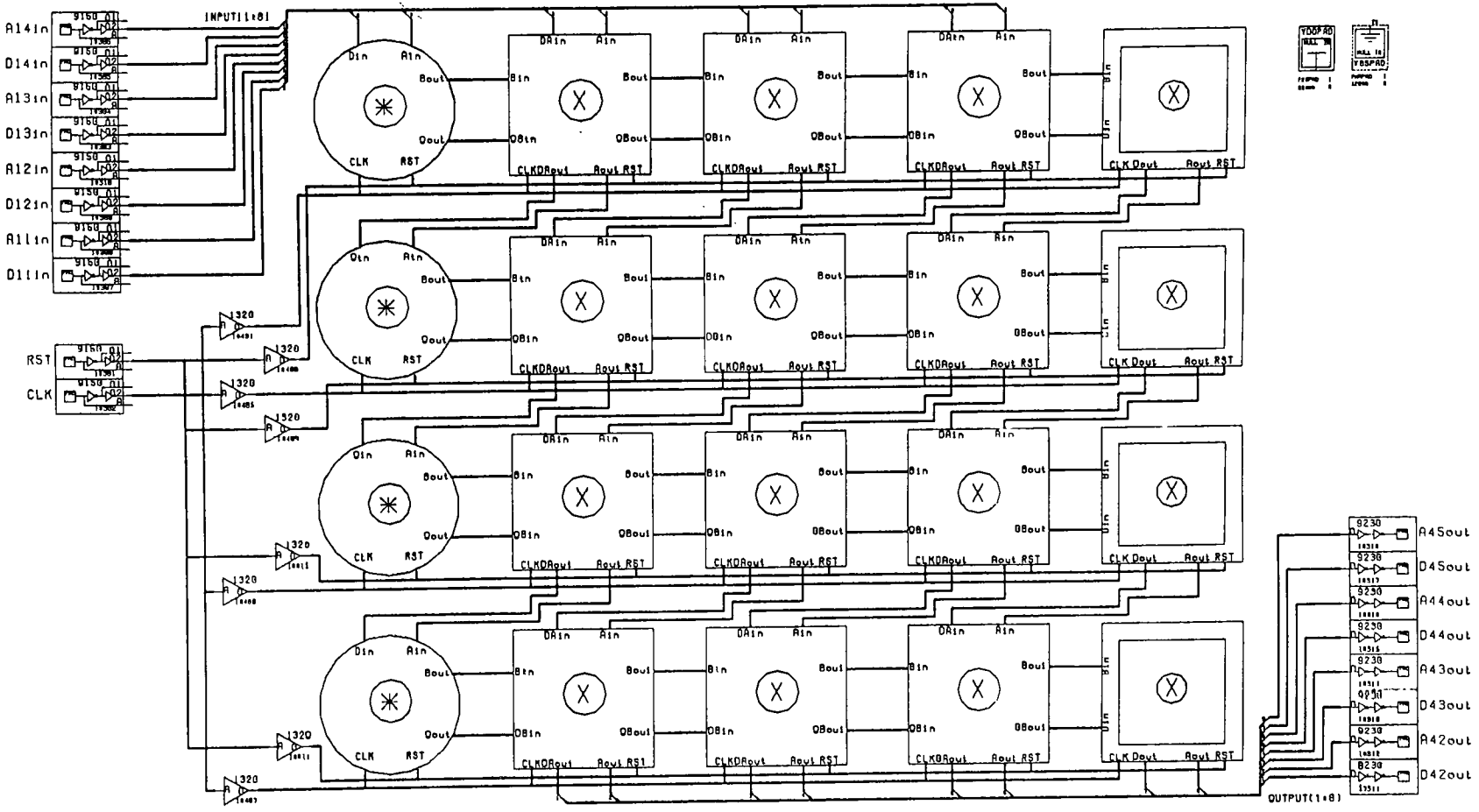


Figure 37. 4x4 Systolic Array using MOSIS3 Standard Cells

11.0 Final Simulation and Testing:

To perform the final simulations and testing, initially Cell Station commands in steps 1 through 5 of figure 33 were completed. The design was converted to use MOSIS3 standard cells, expanded, and checked. During checking it was discovered that additional buffers were needed to drive the clock and reset lines because the fanout was over thirty. Compare figures 26 and 37 for the changes.

The same tests that were run before on the 4x4 array were again completed. These proved to agree exactly. At this point propagation delays were measured for the individual processors. The table below gives the results of the worst case.

Processor	Aout	DAout	Bout	DBout
Circle	--	--	31ns	20ns
Square	31ns	32ns	20ns	20ns
Double Square	32ns	32ns	--	--

Table 5: Propagation Delays for Processors

These values were determined using Quicksim's ability to list times at which a signal changes state.

11.1 Fanout delay:

Since the array was logically functioning properly, the fanout delays were added next, using steps 6 to 7 of figure 33. The clock period remained at 1000ns to avoid any problems with propagation delays and rise and fall times. The previously mentioned delays and times were measured to ensure no problems were actually occuring. The propagation delays are summaraized below.

Processor	Aout	DAout	Bout	DBout
Circle	--	--	31.6ns	20.7ns
Square	31.7ns	34ns	21.1ns	23.9ns
Double Square	33.0ns	34.1ns	--	--

Table 6: Processor Propagation Delays

The above values in table 6 were once again calculate from Quicksim's listing output.

The rise and fall times of the clock signal determined by Cell Station tools were checked during this step and subsequent steps to ensure proper signal timing. The other signals were not critically checked. The typical fanout was three. Whereas the clock signal was connected to at least 20 devices.

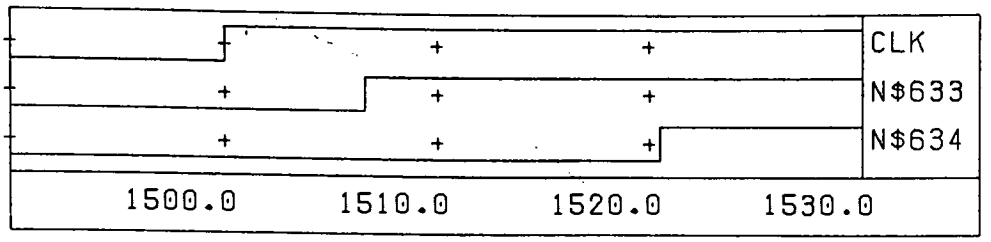


Figure 38. Clock skew after the fanout delay is added (0-1). N$633 is the pad output. N$634 is the buffer output.

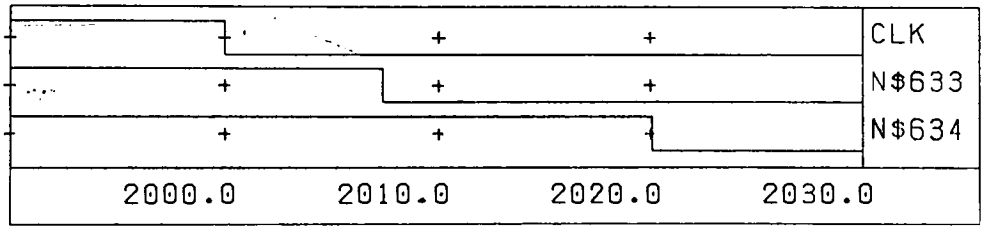


Figure 39. Clock Skew after the fanout delay is added (1-0). N$633 is the pad output. N$634 is the buffer output.

Once the fanout was added, the clock signal, which drove the processors, rise time to change from 4ns to 13.933 and its fall time from 4.19ns to 12.656ns. The clock skew which occurred because of the input pad and additional driving buffer was also considered. A pictorial represntation is given in figures 38 and 39 of the skewing. The input pad delayed the zero to one transistion 6ns. This compounded with the drive buffer produced a total delay of 20.1ns. Similarly the one to zero transition was delayed 20.5ns. All of these restrictions on the clock signal will be considered when the maximum clock frequency is calculated after all delays are added.

A simulation output for a single test case is given in figure 42. The test case is similar to that depicted in figure 27 except that the extra relation 2->4 was added, i.e. 2 is mapped to 4.

11.2 Back Annotated and Wiring Delay:

The standard cells making up the array's integrated circuit were placed and then routed on the chip floor plan. The delays due to the routing wires were back annotated into the simulation properties. These wiring delays were the final ones

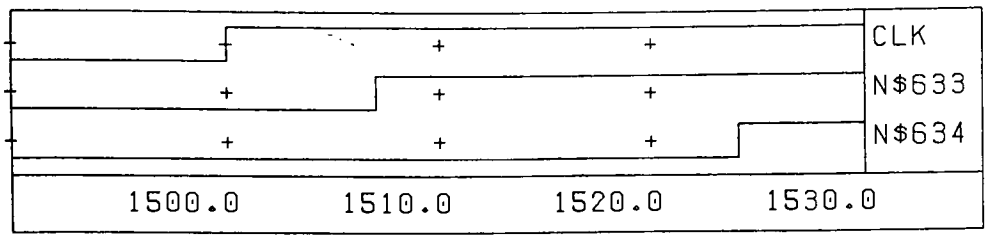


Figure 40. Clock Skew after wiring delay and back annotation are added (0-1). N$633 is the pad output. N$634 is the buffer output.

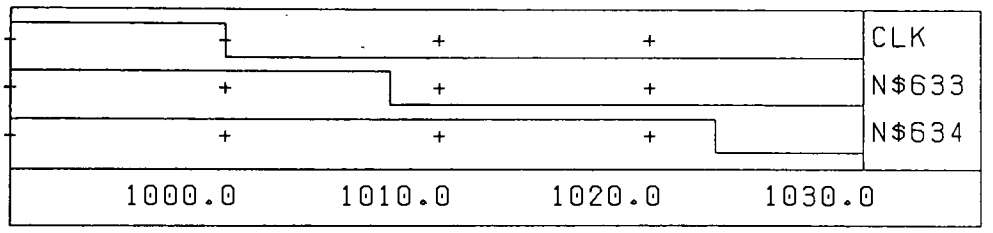


Figure 41. Clock Skew after wiring delay and back annotation are added (1-0). N$633 is the pad output. N$634 is the buffer output.

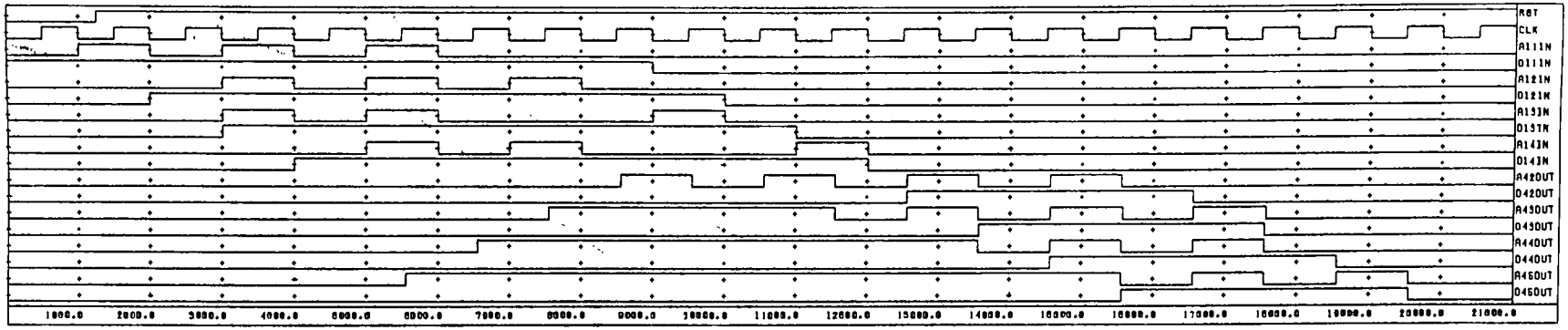


Figure 42. Simulation output for modified test #1 on 4x4 array with fanout delays added.

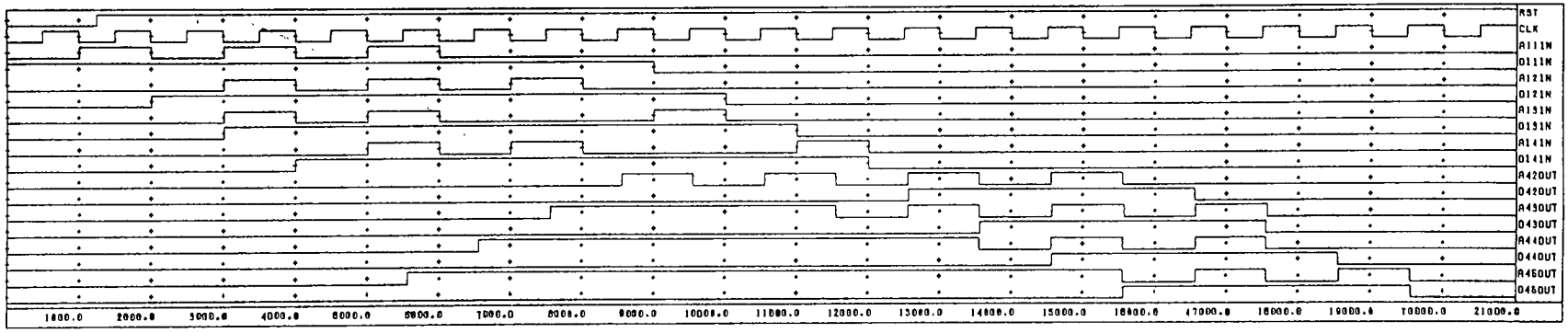


Figure 43. Simulation for modified test #1 on 4x4 array with wiring delays and back annotation added.

to be added for simulation in steps 8 to 18 of figure 33. The clock skew had increased to 22.8ns and 23.8ns for one to zero and zero to one transistions, respectively, see figures 40 to 41. The rise and fall times were also increased to 17.106 and 15.362, respectively. The new propagation delays for the processors are given below.

Processor	Aout	DAout	Bout	DBout
Circle	--	--	31.7ns	21.3ns
Square	32.0ns	34.7ns	21.4ns	21.3ns
Double Square	33.3ns	34.5ns	--	--

Table 7: Final Propagation Delays

Simulation outputs for the three test ran are given in figures 43, 44, and 45 which correspond to the matrices given in figures 27 (with 2->4 added), 29. and 28, respectively. The 1000ns clock period was used during these tests also. In the following section, the minimum clock period will be derived and used.

11.3 Minimum Clock Period:

The minimum clock period needs to assimulate the propagation delays of the processors along with the rise and fall times of

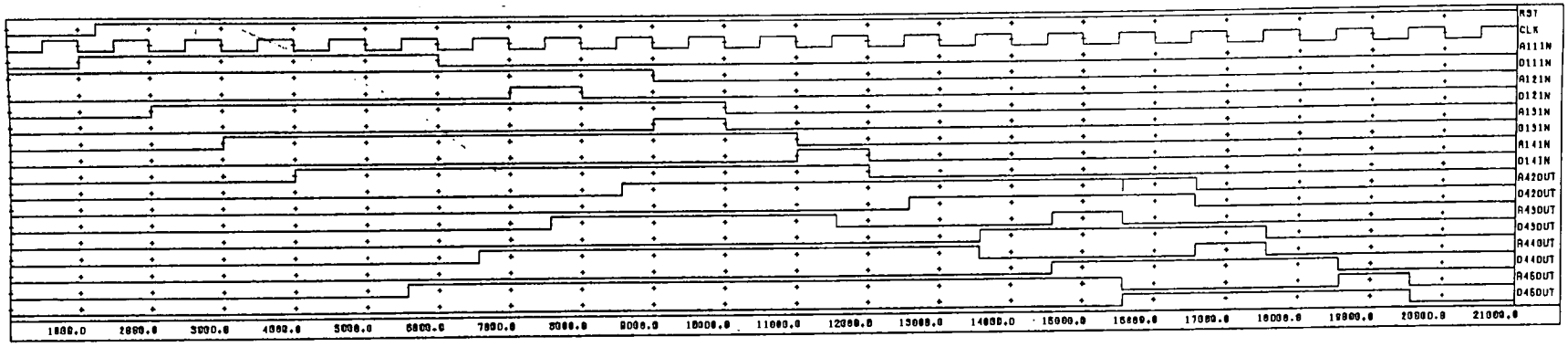


Figure 44. Simulation for test #3 on 4x4 array with wiring delays and back annotation added.

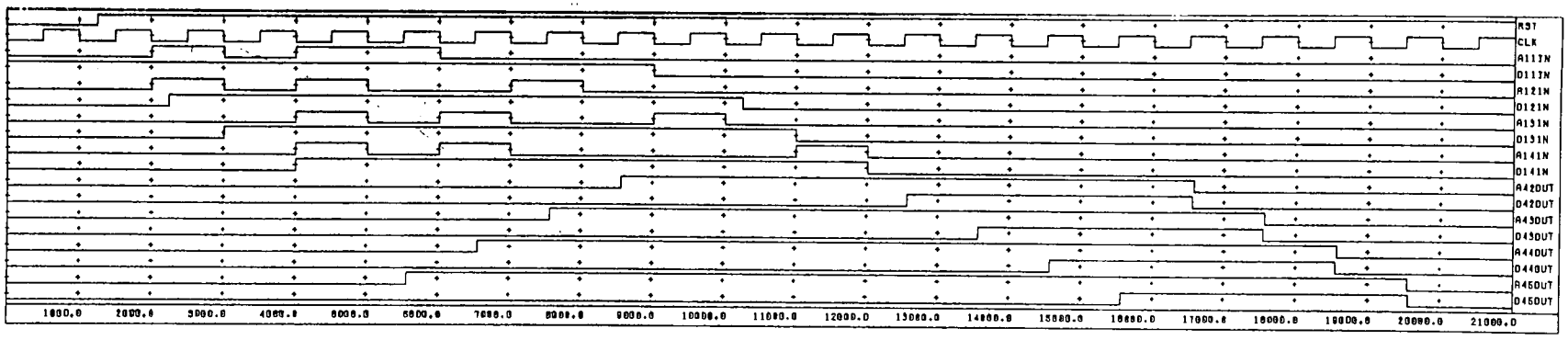


Figure 45. Simulation for test #2 on 4x4 array with wiring delays and back annotation added.

the clock signal. The one state of the clock should be at least 40ns with a fall time of 20ns to completely cover the maximum propagation delay of the processors and the new fall time. Likewise, the zero state should be at least 30ns, minimum allowed is 29ns (Heinbuch 1988), with a 20ns rise time. This would yield a 110ns period, refer to figure 49. These values will ensure the proper operation of the array. Simulations were run with a period of 120ns, the zero state was increased to 40ns, see figures 46 to 48.

11.4 Data Timing Requirements:

The clock signal has been skewed by the input pad and drive buffer. This requires the data to be placed on the lines at a certain period. From examination of figure 49, one can see that the best place is 20ns after the zero to one transition.

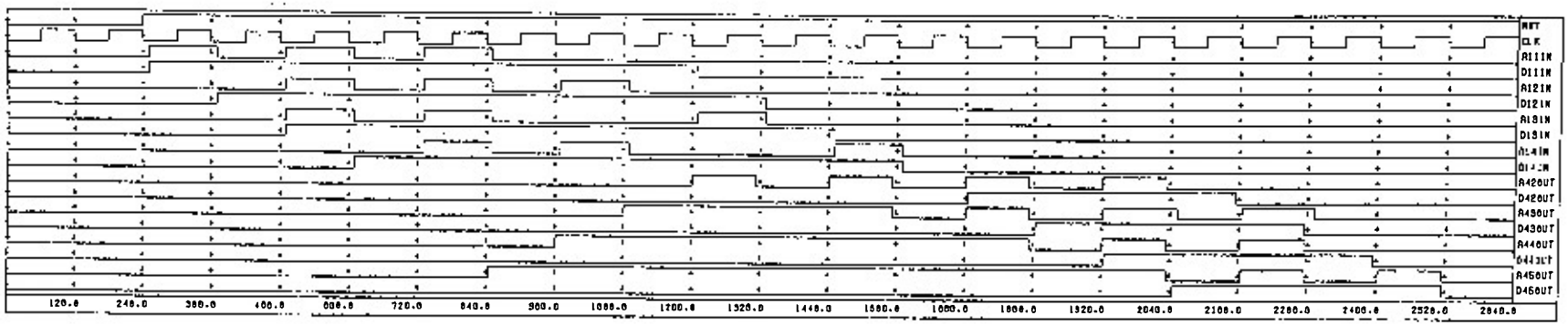


Figure 46. Simulation for modified test #1 on 4x4 array at maximum frequency (8.3 MHz)

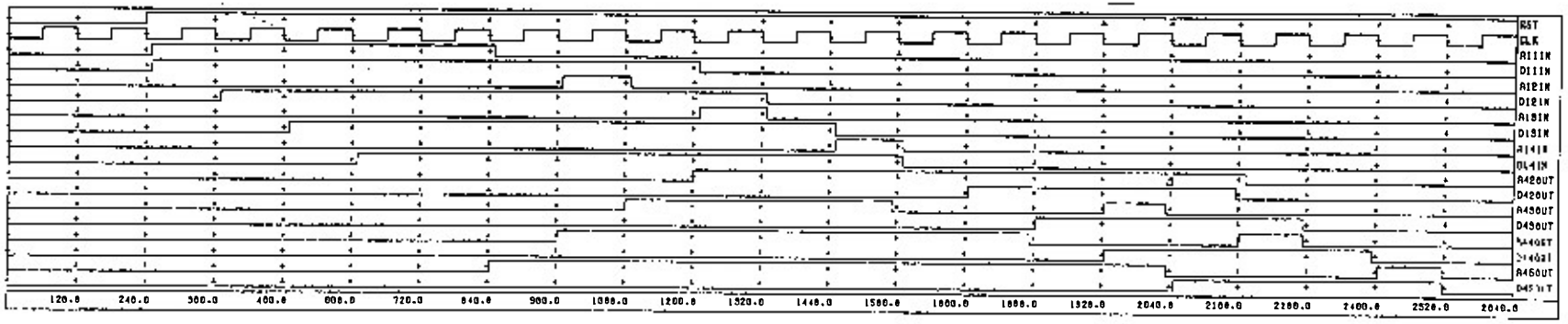


Figure 47. Simulation for test #3 on 4x4 array at maximum frequency (8.3 MHz)

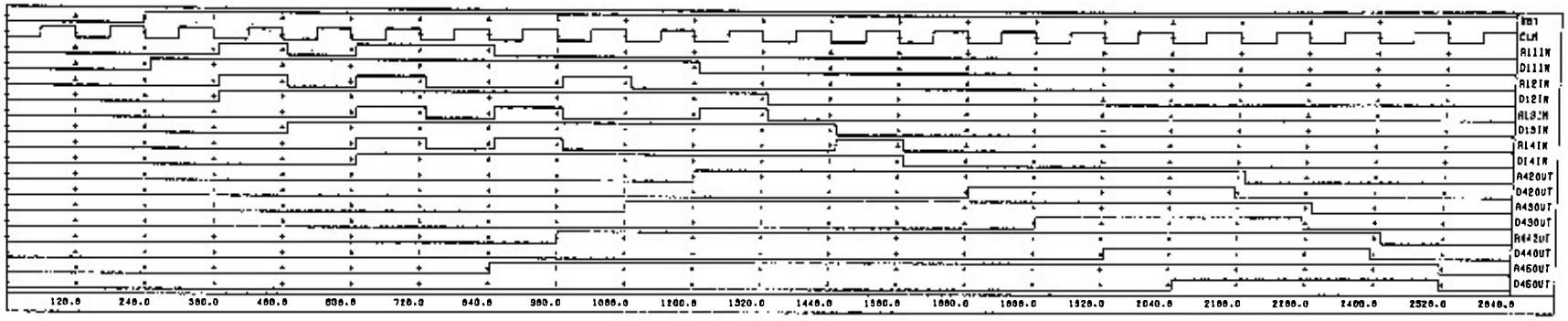


Figure 48. Simulation output for test #2 on 4x4 array at maximum frequency (8.3 MHz)

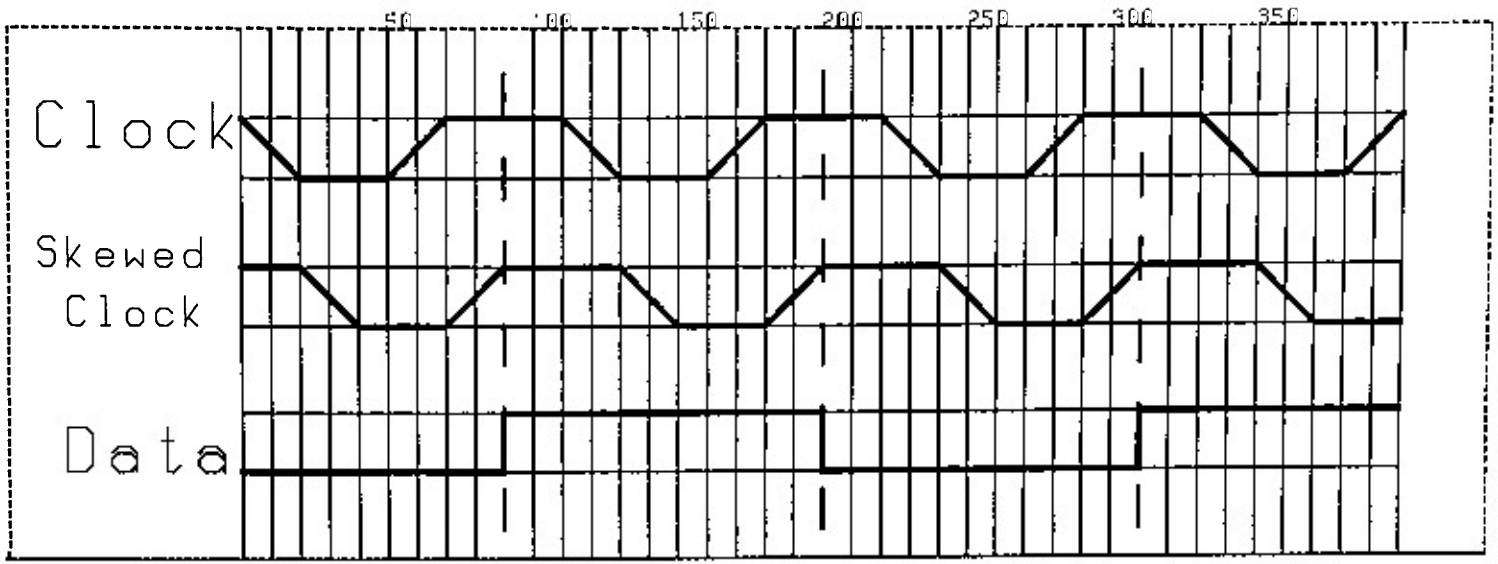


Figure 49. Clock wave form after skewing

12.0 GDS2 OUTPUT:

After fully simulating the systolic array and verifying proper functioning, the final steps were taken to obtain a file to have the chip fabricated. Steps 19 through 22 of figure 33 were completed producing the chip layout in figure 50 and the GDS2_OUPUT file for fabrication.

13.0 Future Endeavors:

There are several more steps which could be completed on this thesis.

13.1 Fabrication and Testing:

Ideally the chip would be fabricated and tested. The test vectors that could be used are those that have already been presented along with matrices of all 0's and all 1's. These additional matrices would test if the R registers are stuck at either 1 or 0.

13.2 Three Chip Design:

The systolic array could be expanded to use three chips with

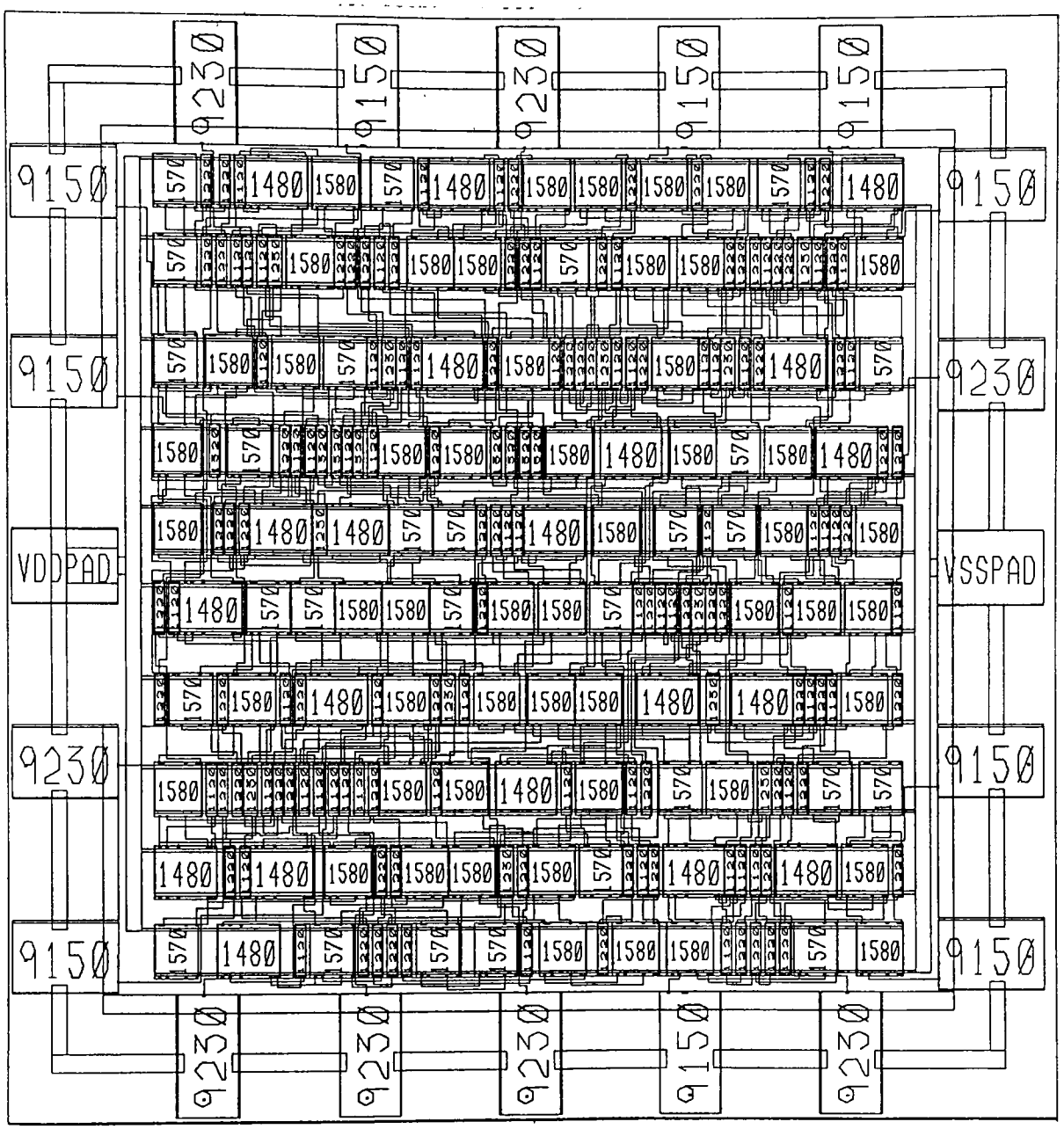


Figure 50. Chip layout of 4x4 systolic array

each chip enclosing a 4x4 section of a larger array. This would allow any array size to be built up. The first chip would have all the processors except for the right most, double square processors. The second chip will be an array of only square processors. The third chip will be an array with the leftmost circle processors removed.

14.0 Conclusion:

Starting from the theory of the Algebraic Path Problem, this thesis has presented a general algorithm by Robert and Trystram (1986) for its solution. The approach and operation of the algorithm was explained and clarified which led to the discussion of the specific systolic array implementation. From this systolic array, a single instance for the APP, the transistive and reflexive closure of a binary relation, was designed and laid out for a CMOS3 standard cell chip design. A minimum clock period of 120ns or maximum frequency of 8.3MHz was determined. The chip is ready to be fabricated, tested and used.

REFERENCES

BENAINI A., ROBERT Y., TOURANCHEAU B. 1989, A New Systolic Architecture for the Algebraic Path Problem, Proc. International Conference on Systolic Arrays, Killarney, Co. Kerry, Ireland, in Systolic Array Processors, ed. McCanny J., McWhirter J., Swartzlander Jr. E., Prentice Hall, 1989, pp. 73 - 82.

GONDRAN M., MINOUX M., VAJDA S. 1984, Graphs and Algorithms, John Wiley & Sons, New York, 1984.

HARTNETT W. 1963, Principles of Modern Mathematics, Harper & Row, New York, 1963.

HEINBUCH D. 1988 ed., CMOS3 Cell Library, Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

KUICH W., SALOMMAA A. 1986, Semiring, Automata, Languages, Springer-Verlag, Berlin, Germany, 1986.

KUNG S. Y. 1988, VLSI Array Processors, Prentice Hall, New Jersey, 1988.

ROBERT Y. 1987, Systolic Algorithms and Architectures, in Automata Networks in Computer Science (Theory and Applications), ed. Soulie F. F., Robert Y., Tchuente M., Princeton University Press, Princeton, New Jersey, 1987, pp. 187 - 228.

ROBERT Y., TRYSTRAM D. 1986, Systolic Solution of the Algebraic Path Problem, Proc. First International Workshop on Systolic Arrays, Oxford, 2-4 July 1986, in Systolic Arrays, ed. Moore W., McCabe A., Urquhart R., Adam Hilger, 1987, pp. 171 - 180.

ROTE G. 1985, A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths; Matrix Inversion), Computing 34 191-219.

Computer Program to Simulate Systolic Array

```
1   Print%=0                :REMark Flag to output to printer
2   StepThrough%=0          :REMark Flag to print individual steps
3   Algo%=0                 :REMark Flag to pick which algorithm
4                           :REMark 0 circuit, 1 program implem.
5   Size%=4                 :REMark Array size
6   DATA 1,1,1,1            :REMark array
7   DATA 0,0,0,0
8   DATA 0,0,0,0
9   DATA 1,0,0,0
10                          :REMark Initialization
11  DIM Input%(Size%-1,Size%-1), Ain%(Size%-1,Size%),
        Aout%(Size%-1,Size%), Bin%(Size%-1,Size%),
        Bout%(Size%-1,Size%), R%(Size%-1,Size%),
        Init%(Size%-1,Size%), DataStreamIn%(3*Size%-2,Size%-1),
        DataStreamOut%(2*Size%-2,Size%-1), II%(Size%-1,Size%-1)
12  DIM DAin%(Size%-1,Size%), DAout%(Size%-1,Size%),
        DBin%(Size%-1,Size%), DBout%(Size%-1,Size%)
13  IF Print% THEN OPEN#3,prt
14  RESTORE 6
15                      :REMark Setup matrix input stream
16  FOR i=0 TO Size%-1
17     FOR j=0 TO Size%-1
18        IF i=j THEN
19           II%(i,j)=1
20        ELSE
21           II%(i,j)=0
22        END IF
23        READ temp%
24        Input%(i,j)=temp%
25     END FOR j
26  END FOR i
27  SetUpDataStream
28  PrintDataStreamIn
29  Systolic
30  PrintDataStreamout
31  IF Print% THEN CLOSE#3
32  STOP                    :REMark Program end
33  :
34  : REMark Sets up the input data stream
35  :
36  DEFine PROCedure SetUpDataStream
37  LOCal i,j
38  :        REMark Insert NIL's
39      FOR i=0 TO Size%-2
40        FOR j=0 TO Size%-1
41           DataStreamIn%(i,j)=-1
```

```
42              DataStreamIn%(2*Size%+i,j)=-1
43          END FOR j
44       END FOR i
45    :        REMark  Insert Data & I matrix
46       FOR i=0 TO Size%-1
47          FOR j=0 TO Size%-1
48             DataStreamIn%(i+j,j)=Input%(j,i)
49             DataStreamIn%(i+j+Size%,j)=II%(i,j)
50          END FOR j
51       END FOR i
52    END DEFine SetUpDataStream
53    :
54    :
55    DEFine PROCedure PrintDataStreamIn
56    LOCal i,j
57       CLS#1
58       FOR i=0 TO 3*Size%-2
59          FOR j=0 TO Size%-1
60             IF Print% THEN PRINT#3,TO j*3;DataStreamIn%(i,j);
61             AT#1,i,j*3:PRINT #1,DataStreamIn%(i,j)
62          END FOR j
63          IF Print% THEN PRINT#3
64       END FOR i
65       IF Print% THEN PRINT#3,\\\\
66    END DEFine PrintDataStreamIn
67    :
68    :
69    DEFine PROCedure PrintDataStreamout
70    LOCal i,j
71       CLS#1
72       FOR i=0 TO 2*Size%-2
73          FOR j=0 TO Size%-1
74             IF Print% THEN PRINT#3,TO j*3;DataStreamOut%(i,j);
75             AT#1,i,j*3:PRINT #1,DataStreamOut%(i,j)
76          END FOR j
77          IF Print% THEN PRINT#3
78       END FOR i
79    END DEFine PrintDataStreamout
80    :
81    : REMark Performs the complete array computations
82    :
83    DEFine PROCedure Systolic
84    LOCal Time%,i,j
85       Time%=0
86       SetUpCells
87       REPeat Time%
88          TransferOutIn Time%
89          IF StepThrough% THEN
```

```
90            DisplayArray
91            PAUSE
92         END IF
93         FOR i=0 TO Size%-1
94             FOR j=0 TO Size%
95               UpdateCell i,j
96            END FOR j
97         END FOR i
98         Time%=Time%+1
99         IF Time%>5*Size%-2 THEN EXIT Time%
100     END REPeat Time%
101  END DEFine Systolic
102  :
103  :
104  DEFine PROCedure TransferOutIn(T%)
105  LOCal i,j
106     FOR j=0 TO Size%
107        IF j<>Size% THEN
108           IF T%<=3*Size%-2 THEN
109              IF DataStreamIn%(T%,j)=-1 THEN
110                 Ain%(0,j)=0:DAin%(0,j)=0
111              ELSE
112                 Ain%(0,j)=DataStreamIn%(T%,j):DAin%(0,j)=1
113              END IF
114           ELSE
115              Ain%(0,j)=0:DAin%(0,j)=0
116           END IF
117        END IF
118        IF j THEN
119           Bin%(0,j)=Bout%(0,j-1)
120           DBin%(0,j)=DBout%(0,j-1)
121        END IF
122     END FOR j
123     FOR i=1 TO Size%-2
124        FOR j=0 TO Size%
125           IF j<>Size% THEN
126              Ain%(i,j) =Aout%(i-1,j+1)
127              DAin%(i,j)=DAout%(i-1,j+1)
128           END IF
129           IF j THEN
130              Bin%(i,j) =Bout%(i,j-1)
131              DBin%(i,j)=DBout%(i,j-1)
132           END IF
133        END FOR j
134     END FOR i
135     FOR j=0 TO Size%
136        IF j<>Size% THEN
137           Ain%(Size%-1,j) =Aout%(Size%-2,j+1)
```

```
138           DAin%(Size%-1,j)=DAout%(Size%-2,j+1)
139        END IF
140        IF j THEN
141           Bin%(Size%-1,j) =Bout%(Size%-1,j-1)
142           DBin%(Size%-1,j)=DBout%(Size%-1,j-1)
143           IF T%>=3*Size% AND T%<=5*Size%-2
144              DataStreamOut%(T%-3*Size%,j-1) =
                   Aout%(Size%-1,j)*(DAout%(Size%-1,j)=1) -
                   (DAout%(Size%-1,j)=0)
145           END IF
146        END IF
147     END FOR j
148 END DEFine TransferOutIn
149 :
150 :
151 DEFine PROCedure SetUpCells
152 LOCal i,j
153     FOR i=0 TO Size%-1
154        FOR j=0 TO Size%
155           Ain%(i,j) =0 :Aout%(i,j) =0
156           DAin%(i,j)=0 :DAout%(i,j)=0
157           Bin%(i,j) =0 :Bout%(i,j) =0
158           DBin%(i,j)=0 :DBout%(i,j)=0
159           Init%(i,j)=1 :R%(i,j)    =0
160        END FOR j
161     END FOR i
162 END DEFine SetUpCells
163 :
164 :
165 DEFine PROCedure UpdateCell(X%,Y%)
166 LOCal y
167     y=Y%
168     SELect ON y
169        ON y=0
170          Circles X%,Y%
171        ON y=1 TO Size%-1
172          Square X%,Y%
173        ON y=Size%
174          Dsquare X%,Y%
175     END SELect
176 END DEFine UpdateCell
177 :
178 :
179 DEFine PROCedure Circles(X%,Y%)
180     IF Algo% THEN
181        IF DAin%(X%,Y%) THEN
182           IF Init%(X%,Y%) THEN
183              Bout%(X%,Y%)=1
```

```
184                DBout%(X%,Y%)=1
185                Init%(X%,Y%)=0
186             ELSE
187                Bout%(X%,Y%)=Ain%(X%,Y%)
188                DBout%(X%,Y%)=1
189             END IF
190          ELSE
191             DAout%(X%,Y%)=0:DBout%(X%,Y%)=0
192          END IF
193       ELSE
194          Bout%(X%,Y%)  =Ain%(X%,Y%)||Init%(X%,Y%)
195          DBout%(X%,Y%)=DAin%(X%,Y%)
196          Init%(X%,Y%)  =NOT(DAin%(X%,Y%))&&Init%(X%,Y%)
197       END IF
198 END DEFine Circles
199 :
200 :
201 DEFine PROCedure Dsquare(X%,Y%)
202       IF Algo% THEN
203          IF DBin%(X%,Y%) THEN
204             IF Init%(X%,Y%) THEN
205                R%(X%,Y%)=Bin%(X%,Y%)
206                Init%(X%,Y%)=0
207                Aout%(X%,Y%)=0
208                DAout%(X%,Y%)=0
209             ELSE
210                Aout%(X%,Y%)=R%(X%,Y%)&&Bin%(X%,Y%)
211                DAout%(X%,Y%)=1
212             END IF
213          ELSE
214             DAout%(X%,Y%)=0:DBout%(X%,Y%)=0
215          END IF
216       ELSE
217          Aout%(X%,Y%)  =Bin%(X%,Y%)&&R%(X%,Y%)
218          DAout%(X%,Y%)=NOT(Init%(X%,Y%))&&DBin%(X%,Y%)
219          R%(X%,Y%)     =(Init%(X%,Y%)&&DBin%(X%,Y%)) ||
                            (NOT(Init%(X%,Y%))&&R%(X%,Y%))
220          Init%(X%,Y%)  =Init%(X%,Y%)&&NOT(DBin%(X%,Y%))
221       END IF
222 END DEFine Dsquare
223 DEFine PROCedure Square(X%,Y%)
224       IF Algo% THEN
225          IF DBin%(X%,Y%) AND DAin%(X%,Y%) THEN
226             IF Init%(X%,Y%) THEN
227                R%(X%,Y%)=Bin%(X%,Y%) && Ain%(X%,Y%)
228                Init%(X%,Y%)=0
229                Aout%(X%,Y%)=0:DAout%(X%,Y%)=0
230                Bout%(X%,Y%)=Bin%(X%,Y%):DBout%(X%,Y%)=1
```

```
231            ELSE
232               Aout%(X%,Y%) = Ain%(X%,Y%) || R%(X%,Y%) &&
                                  Bin%(X%,Y%)
233               DAout%(X%,Y%)=1
234               Bout%(X%,Y%)=Bin%(X%,Y%):DBout%(X%,Y%)=1
235            END IF
236         ELSE
237            DBout%(X%,Y%)=0:DAout%(X%,Y%)=0
238         END IF
239      ELSE
240         Aout%(X%,Y%)=(Bin%(X%,Y%)&&R%(X%,Y%))||Ain%(X%,Y%)
241         DAout%(X%,Y%)=NOT(Init%(X%,Y%))&&DAin%(X%,Y%)
242         DBout%(X%,Y%)=DBin%(X%,Y%)
243         Bout%(X%,Y%)=Bin%(X%,Y%)
244         R%(X%,Y%) = (Init%(X%,Y%) && DBin%(X%,Y%) &&
                         DAin%(X%,Y%) && Ain%(X%,Y%) &&
                         Bin%(X%,Y%)) || (NOT(Init%(X%,Y%)) &&
                         R%(X%,Y%))
245         Init%(X%,Y%) = Init%(X%,Y%) && (NOT(DBin%(X%,Y%)) ||
                           NOT(DAin%(X%,Y%)))
246      END IF
247   END DEFine Square
248   :
249   :
250   DEFine PROCedure DisplayArray
251   LOCal i,j
252      CLS#1
253      FOR j=0 TO Size%-1
254         INK#1,4+3*DAin%(0,j)
255         AT#1,0,10*j:PRINT#1,'A:';Ain%(0,j);DAin%(0,j)
256      END FOR j
257      FOR i=0 TO Size%-1
258         FOR j=0 TO Size%
259            INK#1,4+3*DAout%(i,j)
260            AT#1,5*i+2,10*j :
               PRINT#1,'A:';Aout%(i,j);DAout%(i,j)
261            INK#1,4+3*DBout%(i,j)
262            AT#1,5*i+3,10*j:
               PRINT#1,'B:';Bout%(i,j);DBout%(i,j)
263            INK#1,7
264            AT#1,5*i+4,10*j:PRINT#1,'R:';R%(i,j)
265            INK#1,4+3*Init%(i,j)
266            AT#1,5*i+5,10*j:PRINT#1,'I:';Init%(i,j)
267         END FOR j
268      END FOR i
269      FOR j=1 TO Size%
270         INK#1,4+3*DAout%(Size%-1,j)
271         AT#1,5*(Size%)+3,10*j:
```

```
          PRINT#1,'A:';Aout%(Size%-1,j);DAout%(Size%-1,j)
272     END FOR j
273     INK#1,7
274     PRINT#1,\'Time: ';Time%
275 END DEFine DisplayArray
```